Developers' Perception of Co-Change Patterns: An Empirical Study

Luciana L. Silva*†, Marco Tulio Valente*, Marcelo Maia‡ and Nicolas Anquetil§
*Department of Computer Science, Federal University of Minas Gerais, Brazil - {luciana.lourdes,mtov}@dcc.ufmg.br
†Federal Institute of Triangulo Mineiro, Brazil
‡Faculty of Computing, Federal University of Uberlandia, Brazil - marcmaia@facom.ufu.br
§RMoD Project-Team - INRIA Lille Nord Europe - nicolas.anquetil@inria.fr

***Abstract*—Co-change clusters are groups of classes that frequently change together. They are proposed as an alternative modular view, which can be used to assess the traditional decomposition of systems in packages. To investigate developer's perception of co-change clusters, we report in this paper a study with experts on six systems, implemented in two languages. We mine 102 co-change clusters from the version history of such systems, which are classified in three patterns regarding their projection to the package structure: Encapsulated, Crosscutting, and Octopus. We then collect the perception of expert developers on such clusters, aiming to ask two central questions: (a) what concerns and changes are captured by the extracted clusters? (b) do the extracted clusters reveal design anomalies? We conclude that Encapsulated Clusters are often viewed as healthy designs and that Crosscutting Clusters tend to be associated to design anomalies. Octopus Clusters are normally associated to expected class distributions, which are not easy to implement in an encapsulated way, according to the interviewed developers.**

I. Introduction

In his seminal paper on modularity and information hiding, Parnas developed the principle that modules should hide "difficult design decisions or design decisions which are likely to change" [1]. Nonetheless, Parnas' criteria to decompose systems into modules are not widely used to assess whether—after years of maintenance and evolution—the modules of a system were indeed able to confine changes. In other words, developers typically do not evaluate modular designs using historical data on software changes. Instead, modularity is evaluated most of the times under a structural perspective, using static measures of size, coupling, cohesion, etc [2]–[4]. Less frequently, semantic relations, normally extracted from source code vocabularies, are used [5]–[8].

In previous work [9], [10], we proposed the Co-Change Clustering technique to assess modularity using the history of software changes, widely available nowadays from version control repositories. Co-change clusters are sets of classes that frequently changed together in the past. They are computed applying a cluster algorithm over a co-change graph, which is a graph that represents the co-change relations in a system [11], [12]. A co-change relation between two classes is enabled whenever they are changed by the same commit transaction [13]. We also proposed the usage of distribution maps—a well-known software visualization technique [14]—to reason on co-change patterns, which are recurrent projections of co-change clusters over package structures. However, we did not evaluate to what extent co-change clusters reflect well-defined concerns, according to software developers. We also did not evaluate whether they are useful instruments to detect design anomalies, specially when the clusters crosscut the package structure or have most classes in one package and very few ones in other packages.

To reveal developers' view on the usage of Co-Change Clustering, we report in this paper an empirical study with seven experts on six systems, including one closed-source and large information system implemented in Java and five open-source software tools implemented in Pharo (a Smalltalk-like language). We mine 102 co-change clusters from the version histories of such systems, which are then classified in three patterns regarding their projection over the package structure: Encapsulated Clusters (clusters that when projected over the package structure match all co-change classes in such packages), Crosscutting Clusters (clusters whose classes are spread over several packages, touching few classes in each one), and Octopus Clusters (clusters that have most classes in one package and some "tentacles" in other packages). From the initially computed clusters, 53 clusters (52%) are covered by the proposed co-change patterns. We analyze each of these clusters with developers, asking them two overarching questions: (a) what concerns and changes are captured by the cluster? (b) does the cluster reveal design flaws? Our intention with the first question is to evaluate whether co-change clusters capture cohesive concerns that changed frequently during the software evolution. With the second question, we aim to evaluate whether co-change clusters—specially the ones classified as Crosscutting and Octopus clusters—can reveal design (or modularity) flaws.

Our contributions with this paper are twofold. First, we report a new experience on using Co-Change Clustering in a new set of systems, including both a real-world commercial information system implemented in Java and five open-source systems implemented in a second language (Pharo). Second, we report, summarize, and discuss the developer's views on co-change clusters.

We start by summarizing our technique for extracting co-change clusters (Section II) and by defining the co-change patterns considered in this paper (Section III). Then, we present the research method and steps followed in the study (Section IV). Section V reports the developers' view on co-

change clusters and the main findings of our study. Section VI puts in perspective our findings and the lessons learned with the study. Section VII discusses threats to validity and Section IX concludes.

II. Co-Change Clustering

This section presents the method we follow to extract co-change graphs (Section II-A) and then co-change clusters (Section II-B). A detailed description of the steps presented in this section is available in previous work [9], [10].

A. Co-Change Graphs

A co-change graph is an undirected graph $\{V, E\}$, where V is a set of classes and E is a set of edges. An edge connects two classes (vertices) C_i and C_j if there is a transaction (commit) in the Version Control System that contains C_i and C_j, for $i \neq j$. The weight of an edge represents the number of commits including C_i and C_j.

To extract co-change graphs commit data is preprocessed. First, we discard commits that only change artifacts like script files, documentation, configuration files, etc. Because our focus is on co-changes involving classes. We also remove testing classes, because co-changes between tested and testing classes are usually expected and for this reason are less important. We also remove highly scattered commits, i.e., commits that change a massive number of classes. These commits represent very particular maintenance tasks (e.g., a change in comments on license agreements), which are not recurrent.

Second, from the remaining commits, we select the ones whose textual description refers to a valid maintenance task-ID in a tracking system, like Bugzilla, Jira, etc. When the same task-ID is found in multiple commits, we merge such commits, and consider just the merged commits in the co-change graph. However, it is common to have a significant number of commits not linked to issue reports [9], [15]. Therefore, we apply a time window to select the remaining commits. We merge commits by the same developer when they are performed under a given time interval. In this way, we handle the scenario when the developer commits multiple times when performing the same maintenance task. If such commits are not handled considered, we could miss relevant co-change relations.

Finally, co-change graphs are post-processed pruning edges whose weights is less than a given support threshold. The reason is that such edges are not relevant for our purpose of modeling recurrent maintenance tasks.

As described in this section, the pre and post-processing steps—as well as the clustering step discussed in the next section—depend on some thresholds. The concrete threshold values used in this paper are presented in Section IV-C.

B. Co-Change Clusters

Co-change clusters are set of classes in a co-change graph that frequently changed together. Co-change clusters are extracted automatically using a graph clustering algorithm designed to handle sparse graphs, as is typically the case of co-change graphs [9], [11], [12]. More specifically, we use the Chameleon clustering algorithm, which is an agglomerative and hierarchical clustering algorithm recommended to sparse graphs [16]. Chameleon consists of two phases. In the first phase, a sparse graph is extracted from the original graph (a co-change graph in our case) and a graph partitioning algorithm divides the data set (classes) into sets of clusters. In the second phase, an agglomerative hierarchical mining algorithm is applied to merge the clusters retrieved in the first phase. This algorithm maximizes the number of edges within a cluster (internal similarity) and minimizes the number of edges among clusters (external similarity).

Chameleon requires the number of clusters *M* as an input parameter in the first phase. An inappropriate value of *M* may lead to poor clusters. For this reason, we run Chameleon multiple times varying *M*'s value. After each execution, the previous tested value is decremented by one and the clusters smaller than a minimal threshold are discarded. The goal is to focus on groups of classes that may be used as alternative modular views. Therefore, it is not reasonable to consider clusters with a small number of classes.

After pruning the small clusters, a clustering quality function is computed over the remaining clusters, to provide an overall score for the clusters generated by a given M value. This function combines measures of the clusters cohesion (tight clusters) and cluster separation (highly separated clusters). A detailed description of these measures is out of the scope of this paper and is available elsewhere [9], [10].

III. Co-Change Patterns

In this section, we propose three co-change patterns aiming to represent common instances of co-change clusters. The patterns are defined by projecting clusters over the package structure of an object-oriented system, using distribution maps. Distribution maps are a software visualization technique that represents classes as small squares in large rectangles, which represent packages [14]. The color of the classes represent a property; in our specific case, the co-change cluster.

Co-change patterns are defined using two metrics originally proposed for distribution maps: focus and spread. First, *spread* measures how many packages are touched by a cluster q. Second, *focus* measures the degree the classes in a co-change cluster dominate their packages. For example, if a cluster touches all classes of a package, its focus is one. In formal terms, the *focus* of a co-change cluster q is defined as follows:

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

where

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

The measure $touch(q, p_i)$ represents the number of classes in a cluster q located in package p_i divided by the number of classes in p_i that are included in at least one co-change cluster. Similarly, $touch(p_i, q)$ is the number of classes in package p_i that are a member of cluster q divided by the number of classes

in q. Focus ranges between 0 and 1, where 1 means that the cluster entirely dominates the packages it touches.

Using focus and spread, we propose three patterns of co-change clusters, as follows:

Encapsulated: An Encapsulated co-change cluster q dominates all classes of the packages it touches, i.e.,

$$Encapsulated(q), \text{if } focus(q) == 1$$

Figure 1 shows two examples of Encapsulated Clusters.[1] All classes in Cluster 9 (blue) are located in the same package, which only has classes in this cluster. Similarly, Cluster 10 (green) has classes located in three packages. Moreover, these three packages do not have classes in other clusters.

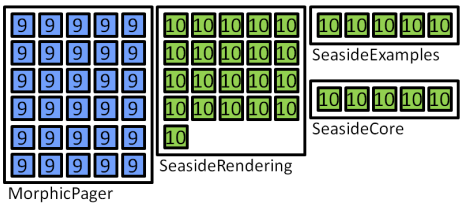

Fig. 1. Encapsulated clusters (Glamour)

Crosscutting: Conceptually, a Crosscutting Cluster is spread over several packages but touches few classes in each one. In practical terms, we propose the following thresholds to represent a Crosscutting cluster q:

$$Crosscutting(q), \text{ if } spread(q) \geq 4 \ \wedge \ focus(q) \leq 0.3$$

Figure 2 shows an example of Crosscutting cluster. Cluster 8 (red) is spread over seven packages, but does not dominate any of them (its focus is 0.14).

Octopus: Conceptually, an Octopus Cluster q has two sub-clusters: a body B and a set of tentacles T. The body has most classes in the cluster and the tentacles have a very low focus, as follows:

$$Octopus(q, B, T) = \text{if} \quad \begin{array}{l} touch(B, q) > 0.50 \wedge \\ focus(T) \leq 0.25 \ \wedge \\ focus(q) > 0.3 \end{array}$$

By requiring $focus(q) > 0.3$, a cluster cannot be classified as Crosscutting and Octopus, simultaneously.

Figure 3 shows an Octopus cluster, whose body has 22 classes, located in one package. The cluster has a single tentacle class. When considered as an independent sub-cluster, this tentacle has focus 0.005. Finally, the whole Octopus has focus 0.78, which avoids its classification as Crosscutting.

As usual in the case of metric-based rules to detect code patterns [17], [18], the proposed strategies to detect co-change patterns depend on thresholds to specify the expected spread and focus values. To define such thresholds we based on our previous experiences with co-change clusters extracted for open-source Java-based systems [9], [10]. Typically, low focus values are smaller than 0.3 and high spread values are greater or equal to four packages.

[1] All examples used in this section are real instances of co-change clusters, extracted from the subject systems used in this paper, see Section IV-B.

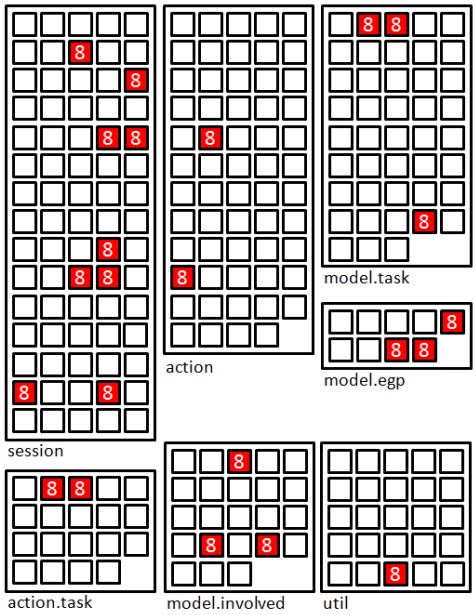

Fig. 2. Crosscutting cluster (SysPol)

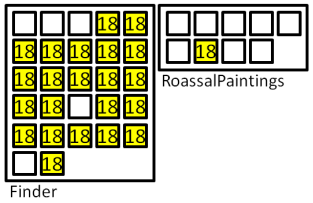

Fig. 3. Octopus cluster (Moose)

IV. Study Design

In this section we present the questions that motivated our research (Section IV-A). We also present the dataset (Section IV-B), the thresholds selection (Section IV-C), the steps we followed to extract the co-change clusters (Section IV-D), and to conduct the interviews (Section IV-E).

A. Research Questions

With this research, our *goal* is to investigate from the point of view of expert developers and architects the concerns represented by co-change patterns. We also evaluate whether these patterns are able to indicate design anomalies, in the *context* of Java and Pharo object-oriented systems. To achieve these goals, we pose three research questions in the paper:

RQ #1: To what extent do the proposed co-change patterns cover real instances of co-change clusters?

RQ #2: How developers describe the clusters matching the proposed co-change patterns?

RQ #3: To what extent do the clusters matching the proposed co-change patterns indicate design anomalies?

With RQ #1, we check whether the proposed strategy to detect co-change patterns match a representative set of co-change clusters. With the second and third RQs we collect and organize the developers' view on co-change patterns. Specifically, with the second RQ we check how developers describe the concerns and requirements implemented by the proposed co-change patterns. With the third RQ, we check whether clusters matching the proposed co-change patterns—specially the ones classified as Crosscutting and Octopus—are usually associated to design anomalies.

B. Target Systems

To answer our research questions, we investigate the following six systems: (a) SysPol, which is a closed-source information system implemented in Java that provides many services related to the automation of forensics and criminal investigation processes; the system is currently used by one of the Brazilian state police forces (we are omitting the real name of this system, due to a non-disclosure agreement with the software organization responsible for SysPol's implementation and maintenance); (b) five open-source systems implemented in Pharo [19], which is a Smalltalk-like language. We evaluate the following Pharo systems: Moose (a platform for software and data analysis), Glamour (an infrastructure for implementing browsers), Epicea (a tool to help developers share untangled commits), Fuel (an object serialization framework), and Seaside (a framework for developing web applications).

Table I describes these systems, including information on number of lines of code (LOC), number of packages (NOP), number of classes (NOC), number of commits extracted for each system, and the time frame considered in this extraction.

TABLE I
TARGET SYSTEMS

System	LOC	NOP	NOC	Commits	Period
SysPol	63,754	38	674	9,072	10/13/2010 - 08/08/2014
Seaside	26,553	28	695	5,741	07/17/2013 - 12/08/2014
Moose	33,967	36	505	2,417	01/21/2013 - 11/17/2014
Fuel	5,407	6	136	2,009	08/05/2013 - 12/03/2014
Epicea	26,260	9	222	1,400	08/15/2013 - 11/15/2014
Glamour	21,076	24	452	3,213	02/08/2013 - 11/27/2014

C. Thresholds Selection

For this evaluation, we use the same thresholds of our previous experience with Co-Change Clustering [9], [10]. We reused the thresholds even for the Pharo systems, because they are decomposed in packages and classes, as in Java.

SysPol is a closed-source system developed under agile development guidelines. In this project, the tasks assigned to the development team usually have an estimated duration of one working day. For this reason, we set up the time window threshold used to merge commits as one day, i.e., commits performed in the same calendar day by the same author are merged. Regarding the Pharo systems, developers have more freedom to select the tasks to work on as common in open-source systems. Moreover, they usually only commit after finishing and testing a task (as explained to us by Pharo's leading software architects). However, Pharo commits are performed per package. For example, a maintenance task that involves changes in classes located in packages P_1 and P_2 is concluded using two different commits: a commit including the classes located in P_1 and another containing the classes in P_2. For this reason, in the case of the five Pharo systems, we set up the time window used to merge commits as equal to one hour. On the one hand, this time interval is enough to capture all commits related to a given maintenance task, according to Pharo architects. On the other hand, developers usually take more than one hour to complete a next task after committing the previous one. Finally, we randomly selected 50 change sets from one of the Pharo systems (Moose) to check manually with one of system's developer. He confirmed that all sets refer to unique programming task.

D. Extracting the Co-Change Clusters

We start by preprocessing the extracted commits to compute co-change graphs. Table II presents four measures: (a) the initial number of commits considered for each system; (b) the number of discard operations targeting commits that do not change classes or change a massive number of classes; (c) the number of merge operations targeting commits referring to the same Task-ID in the tracking system or performed under the time window thresholds; (d) the number of change sets effectively used to compute the co-change graphs. By change sets we refer to the commits used to create the co-change graphs, including the ones produced by the merge operations.

TABLE II
PREPROCESSING FILTERS AND NUMBER OF CO-CHANGE CLUSTERS

System	Commits	Discard Ops	Merge Ops	Change Sets
SysPol	9,072	1,619	1,447	1,951
Seaside	5,741	1,725	1,421	1,602
Moose	2,417	289	762	856
Fuel	2,009	395	267	308
Epicea	1,400	29	411	448
Glamour	3,213	2,722	1,075	1,213

After applying the preprocessing and post-processing filters, we use the ModularityCheck[2] tool to compute the co-change clusters [20]. Table III shows the number of co-change clusters computed for each system (102 clusters, in total).

TABLE III
NUMBER OF CO-CHANGE CLUSTERS PER SYSTEM

System	# clusters	System	# clusters
SysPol	20	Fuel	8
Seaside	25	Epicea	14
Moose	20	Glamour	15

Figure 4 shows the distribution of the densities of the co-change clusters extracted for each system. Density is a key property in co-change clusters, because it assures that there is

[2]http://aserg.labsoft.dcc.ufmg.br/modularitycheck

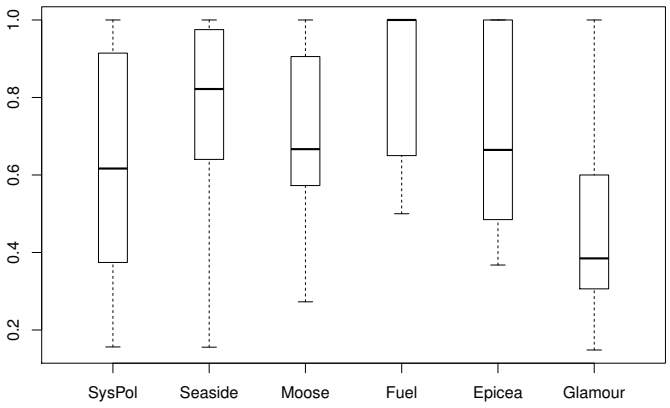

Fig. 4. Co-change clusters density

a high probability of co-changes between each pair of classes in the cluster. The SysPol's clusters have a median density of 0.63, whereas the co-change graph extracted for this system has a density of 0.20. The clusters of the Pharo systems have a median density ranging from 0.39 (Glamour) to 1.00 (Fuel), whereas the highest density of the co-change graphs for these systems is 0.11 (Fuel).

E. Interviews

Table IV describes the number of expert developers we interviewed for each system and how long they have been working on the systems. In total, we interviewed seven experienced developers. In the case of SysPol, we interviewed the lead architect of the system, who manages a team of around 15 developers. For Moose, we interviewed two experts. For the remaining Pharo systems, we interviewed a single developer per system.

TABLE IV
EXPERT DEVELOPERS' PROFILE

System	Developer ID	Experience (Years)	# Emails/Chats
SysPol	D1	2	44
Seaside	D2	5	0
Moose	D3;D4	4.5	6
Fuel	D5	4.5	0
Epicea	D6	2.5	2
Glamour	D7	4	1

We conducted face-to-face and Skype semi-structured interviews with each developer using open-ended questions [21]. During the interviews, we presented all co-change clusters that matched one of the proposed co-change patterns to the developers. We used Grounded Theory [22] to analyze the answers and to organize them into categories and concepts (sub-categories). The interviews were transcribed and took approximately one and half hour (with each developer; both Moose developers were interviewed in the same session). In some cases, we further contacted the developers by e-mail or text chat to clarify particular points in their answers. Table IV also shows the number of clarification mails and chats with each developer. For Moose, we also clarified the role of some classes with its leading architect (D8) who has been working for 10 years in the system.

V. RESULTS

In this section, we present the developer's perceptions on the co-change clusters, collected when answering the proposed research questions.

A. To what extent do the proposed co-change patterns cover real instances of co-change clusters?

To answer this RQ, we categorize the co-change clusters as Encapsulated, Crosscutting, and Octopus, using the definitions proposed in Section III. The results are summarized in Table V. As we can observe, the proposed co-change patterns cover from 35% (Epicea) to 72% (Seaside) of the clusters extracted for our subject systems. We found instances of Octopus Clusters in all six systems. Instances of Encapsulated Clusters are found in all systems, with the exception of SysPol. By contrast, Crosscutting Clusters are less common. In the case of four systems (Seaside, Moose, Fuel, and Epicea) we did not find a single instance of this pattern.

TABLE V
NUMBER AND PERCENTAGE OF CO-CHANGE PATTERNS

System	Encapsulated	Crosscutting	Octopus	Total
SysPol	0 (0%)	8 (40%)	5 (25%)	13 (65%)
Seaside	7 (28%)	0 (0%)	11 (44%)	18 (72%)
Moose	5 (25%)	0 (0%)	1 (5%)	6 (30%)
Fuel	3 (37%)	0 (0%)	1 (13%)	4 (50%)
Epicea	3 (21%)	0 (0%)	2 (14%)	5 (35%)
Glamour	4 (27%)	1 (7%)	2 (20%)	7 (47%)
Total	22 (41.5%)	9 (17%)	22 (41.5%)	53 (52%)

In summary, we found 53 co-change clusters matching one of the proposed co-change patterns (52%). The remaining clusters do not match the proposed patterns because their spread and focus do not follow the thresholds defined in Section III.

B. How developers describe the clusters matching the proposed co-change patterns?

To answer this RQ, we presented each cluster categorized as Encapsulated, Crosscutting, or Octopus to the developer of the respective system and asked him to describe the central concerns implemented by the classes in these clusters.

Encapsulated Clusters: The codes extracted from developers' answers for clusters classified as Encapsulated are summarized in Table VI. The table also presents the package that encapsulates each cluster. The developers easily provided a description for 21 out of 22 Encapsulated clusters. A cluster encapsulated in the `Core` package of Moose (Cluster 16) is the only one the developers were not able to describe by analyzing only the class names. Therefore, in this case we asked the experts to inspect the commits responsible to this cluster and they concluded that the co-change relations are due to *"several small refactorings applied together"*. Since these refactorings are restricted to classes in a single package, they were not filtered out by the threshold proposed to handle highly scattered commits.

Analyzing the developers' answers, we concluded that all clusters in Table VI include classes that implement clear and

TABLE VI
CONCERNS IMPLEMENTED BY ENCAPSULATED CLUSTERS

System	Cluster	Packages	Codes
Seaside	1	Pharo20ToolsWeb	Classes to compute information such as memory and space status
	2	ToolsWeb	Page to administrate Seaside applications
	3	Pharo20Core	URL and XML enconding concerns
	4	Security, PharoSecurity	Classes to configure security strategies
	5	JSONCore	JSON renderer
	6	JavascriptCore	Implementation of JavaScript properties in all dialects
	7	JQueryCore	JQuery wrapper
Glamour	8	MorphicBrick	Basic widgets for increasing performance
	9	MorphicPager	Glamour browser
	10	SeasideRendering, SeasideExamples, SeasideCore	Web renderer implementation
	11	GTInspector	Object inspector implementation
Moose	12	DistributionMap	Classes to draw distribution maps
	13	DevelopmentTools	Scripts to use Moose in command line
	14	MultiDimensionsDistributionMap	Distribution map with more than one variable
	15	MonticelloImporter	Monticello VCS importer
	16	Core	Several small refactoring applied together
Fuel	17	Fuel	Serializer and materializer operations
	18	FuelProgressUpdate	Classes that show a progress update bar
	19	FuelDebug	Implementation of the main features of the package
Epicea	20	Hiedra	Classes to create vertices and link them in a graph
	21	Mend	Command design pattern for modeling change operations
	22	Xylem	Diff operations to transform a dictionary X into Y

well-defined concerns. For this reason, we classify all clusters in a single category, called *Specific Concerns*. For example, in Seaside, Cluster 4 has classes located in two packages: `Security` and `PharoSecurity`. As indicated by their names, these two packages are directly related to security concerns. In Glamour, Cluster 10 represents planned interactions among Glamour's modules, as described by Glamour's developer:

"The `Rendering` *package has web widgets and rendering logic. The* `Presenter` *classes in the* `Rendering` *package represent an abstract description for a widget, which is translated into a concrete widget by the* `Renderer`*. Thus, when the underlying widget library (*`Core`*) is changed, the* `Renderer` *logic is also changed. After that, the* `Examples` *classes have to be updated."* (D7)

Crosscutting Clusters: Table VII presents the codes extracted for the Crosscutting Clusters detected in SysPol, which concentrates 8 out of 9 Crosscutting Clusters considered in our study. We identified that these Crosscutting Clusters usually represent *Assorted Concerns* (category) extracted from the following common concepts:

Assorted, Mostly Functional Concerns. In Table VII, 7 out of 8 Crosscutting Clusters express SysPol's functional concerns. Specifically, four clusters are described as a collection of several concerns (the reference to several is underlined, in Table VII). In the case of these clusters, the classes in a package tend to implement multiple concerns; and a concern tend to be implemented by more than one package. For example, SysPol's developer made the following comments when analyzing one of the clusters:

"`CreateArticle` *is a quite generic use case in our system. It is usually imported by other use cases. Sometimes, when implementing a new use case, you must first change classes associated to* `CreateArticle`*"* (D1, on Cluster 2)

"These classes represent a big module that supports `Task` *related features and that contain several use cases. Classes related to* `Agenda` *can change with* `Task` *because there are* `Tasks` *that can be saved in a* `Agenda`*"* (D1, on Cluster 4)

We also found a single Crosscutting Cluster in Glamour, which has 12 classes spread across four packages, with focus 0.26. According to Glamour's developer *"These classes represent changes in text rendering requirements that crosscut the rendering engine and their clients"* (D7).

Assorted, Mostly Non-functional Concerns. Cluster 8 is the only one which expresses a non-functional concern, since its classes provide access to databases (and also manipulate `Article` templates).

Therefore, at least in SysPol, we did not find a strong correspondence between recurrent and scattered co-change relations—as captured by Crosscutting Clusters—and classical crosscutting concerns, such as logging, distribution, persistence, security, etc. [23], [24]. This finding does not mean that such crosscutting concerns have a well-modularized implementation in SysPol (e.g., using aspects), but that their code is not changed with frequency.

Octopus Clusters: From the developers' answers, we observed that all Octopus represent *Partially Encapsulated Concerns* (category), as illustrated by the following clusters:[3]

- In Moose, there is an Octopus (see Figure 3) whose body implement browsers associated to Moose panels (`Finder`) and the tentacle is a generic class for visualization, which is used by the `Finder` to display visualizations inside browsers.

[3]Due to the lack of space, we do not report the Octopus clusters' codes. We provide this content at http://aserg.labsoft.dcc.ufmg.br/ICSME15-CoChanges

TABLE VII
CONCERNS IMPLEMENTED BY CROSSCUTTING CLUSTERS IN SYSPOL

ID	Spread	Focus	Size	Codes
1	9	0.26	45	Several concerns, search case, search involved in crime, insert conduct
2	9	0.22	29	Several concerns, seizure of material, search for material, and create article
3	10	0.20	31	Requirement related to the concern *analysis*, including review analysis and analysis in flagrant
4	12	0.15	31	Several classes are associated to the *task* and *consolidation* concerns
5	15	0.29	35	Subjects related to create article and to prepare expert report
6	9	0.22	24	Several concerns in the model layer, such as criminal type and indictment
7	7	0.14	24	Features related to people analysis, insertion, and update
8	4	0.30	12	Access to the database and article template

- In Glamour, there is an Octopus whose body implement a declarative language for constructing browsers and the tentacles are UI widgets. Changes in this language (e.g., to support new types of menus) propagate to the `Renderer` (to support the new menu renderings).

C. To what extent do clusters matching the proposed co-change patterns indicate design anomalies?

We also asked the developers whether the clusters are somehow related to design or modularity anomalies, including bad smells, misplaced classes, architectural violations, etc.

Encapsulated Clusters: In the case of Encapsulated Clusters, design anomalies are reported for a single cluster in Glamour (Cluster 9, encapsulated in the `MorphicPager` package, as reported in Table VI). Glamour's developer made the following comment on this cluster:

"The developer who created this new browser did not follow the guidelines for packages in Glamour. Despite of these classes define clearly the browser creation concern, the class `GLMMorphicPagerRenderer` *should be in the package* `Renderer` *and the class* `GLMPager` *should be in the package* `Browser`*"* (D7, on Cluster 9)

Interestingly, this cluster represents a conflict between structural and logical (or co-change based) coupling. Most of the times, the two mentioned classes changed with classes in the `MorphicPager` package. Therefore, the developer who initially implemented them in this package probably favoured this logical aspect in his decision. However, according to Glamour's developer there is a structural force that is more important in this case: subclasses of `Renderer`, like `GLMMorphicPagerRenderer` should be in their own package; the same for subclasses of `Browser`, like `GLMPager`.

Crosscutting Clusters: SysPol's developer explicitly provided evidences that six Crosscutting clusters (67%) are related to *Design Anomalies* (category), including three kind of problems (concepts):

Low Cohesion/High Coupling (two clusters). For example, Cluster 2 includes a class, which is "one of the classes with the highest coupling in the system." (D1)

High Complexity Concerns (two clusters). For example, Cluster 4 represents "a difficult part to understand in the system and its implementation is quite complex, making it hard to apply maintenance changes." (D1)

Package Decomposition Problems (two clusters). For example, 27 classes in Cluster 1 "include implementation logic that should be in an under layer." (D1)

SysPol's developer also reported reasons for *not* perceiving a design problem in the case of three Crosscutting Clusters (Clusters 6, 7, and 8). According to the developer, these clusters have classes spread over multiple architectural layers (like `Session`, `Action`, `Model`, etc), but implementing operations related to the same use case. According to the developer, since these layers are defined by SysPol's architecture, there is no alternative to implement the use cases without changing these classes.

Octopus Clusters: SysPol's developer provided evidences that two out of five Octopus Clusters in the system are somehow related to design anomalies. The design anomalies associated to Octopus Clusters are due to *Package Decomposition Problems*. Moreover, a single Octopus Cluster among the 17 clusters found in the Pharo tools is linked to this category. For example, in Epicea, one cluster includes "some classes located in the `Epicea` package, which should be moved to the `Ombu` package". It is worth mentioning that these anomalies were unknown to the developers. They were detected after inspecting the clusters to comprehend their concerns.

In contrast, the developers did not find design anomalies in the remaining 16 Octopus clusters detected in the Pharo systems. As an example from Moose, the developer explained as follows the Octopus associated to Cluster 18 (see Figure 3):

"The propagation starts from RoassalPaintings to Finder. Whenever something is added in the RoassalPaintings, it is often connected with adding a menu entry in the Finder." (D8)

Interestingly, the propagation in this case happens from the tentacle to the body classes. It is a new feature added to `RoassalPaintings` that propagates changes to the body classes in the `Finder` package. Because Roassal (a visualization engine) and Moose (a software analysis platform) are different systems, it is more complex to refactor the tentacles of this octopus.

VI. DISCUSSION

In this section, we put in perspective our findings and the lessons learned with the study.

A. Applications on Assessing Modularity

On the one hand, we found that Encapsulated Clusters typically represent well-designed modules. Ideally, the higher

the number of Encapsulated Clusters, the higher the quality of a module decomposition. Interestingly, Encapsulated Clusters are the most common co-change patterns in the Pharo software tools we studied, which are generally developed by high-skilled developers. On the other hand, Crosscutting Clusters in SysPol tend to reveal design anomalies with a precision of 67% (at least in our sample of eight Crosscutting Clusters). Typically, these anomalies are due to concerns implemented using complex class structures, which suffer from design problems like *high coupling/low cohesion*. They are not related to classical non-functional concerns, like logging, persistence, distribution, etc. We emphasize that the differences between Java (SysPol) and Pharo systems should not be associated exclusively to the programming language. For example, in previous studies we evaluated at least two Java-based systems without Crosscutting Clusters [10]. In fact, SysPol's expert associates the modularity problems found in the system to a high turnover in the development team, which is mostly composed by junior developers and undergraduate students.

Finally, developers are usually skeptical about removing the octopus' tentacles, by for example moving their classes to the body by inserting a stable interface between the body and the tentacles. For example, Glamour's developer made the following comments when asked about these possibilities:

"Unfortunately, sometimes it is difficult to localize changes in just one package. Even a well-modularized system is a system after all. Shielding changes in only one package is not absolutely possible." (D7)

B. The Tyranny of the Static Decomposition

Specifically for Crosscutting Clusters, the false positives we found are due to maintenance tasks whose implementation requires changes in multiple layers of the software architecture (like user interface, model, persistence, etc). Interestingly, the expert developers usually view their static software architectures as dominant structures. Changes that crosscut the layers in this architecture are not perceived as problems, but as the only possible implementation solution in face of their current architectural decisions. During the study, we referred to this recurrent observation as the *tyranny of the static decomposition*. We borrowed the term from the "tyranny of the dominant decomposition" [25], normally used in aspect-oriented software development to denote the limitations of traditional languages and modularization mechanisms when handling crosscutting concerns.

In future work, we plan to investigate this tyranny in details, by arguing developers if other architectural styles are not possible, for example centered on domain-driven design principles [26]. We plan to investigate whether information systems architected using such principles are less subjected to crosscutting changes, as the ones we found in SysPol.

C. (Semi-)Automatic Remodularizations

Modular decisions deeply depend on software architects expertise and also on particular domain restrictions. For example, even for SysPol's developer it was difficult to explain and reason about the changes captured by some of the Crosscutting Clusters detected in his system. For this reason, the study did not reveal any insights on techniques or heuristics that could be used to (semi-)automatically remove potential design anomalies associated to Crosscutting Clusters. However, co-change clusters readily meet the concept of virtual separation of concerns [27], [28], which advocates module views that do not require syntactic support in the source code. In this sense, co-change clusters can be an alternative to the Package Explorer, helping developers to comprehend the spatial distribution of changes in software systems.

D. Limitations

We found three co-change clusters that are due to floss refactoring, i.e., programming sessions when the developer intersperses refactoring with other kinds of source code changes, like fixing a bug or implementing a new feature [29]. To tackle this limitation, we can use tools that automatically detect refactorings from version histories, like Ref-Finder [30] and Ref-Detector [31]. Once the refactorings are identified, we can remove co-change relations including classes only modified as prescribed by the identified refactorings.

Furthermore, 49 co-change clusters have no matched the proposed patterns (48%). We inspected them to comprehend why they were not categorized as Encapsulated, Crosscutting, or Octopus. We observed that 32 clusters are well-confined in packages, i.e., they touch a single package but their focus is lower than 1.0. This behavior does not match Encapsulated pattern because these clusters share the packages they touch with other clusters. Moreover, we also identified 14 clusters similar to Octopus, i.e., they have bodies in a package and arms in others. However, some clusters have bodies smaller and others have arms tinier than the threshold settings. The remaining three clusters touch very few classes per package but their spread is lower than the threshold settings.

Finally, we did not look for false negatives, i.e., sets of classes that changed together, but that are not classified as co-change clusters by the Chameleon graph clustering algorithm. Usually, computing false negatives in the context of architectural analysis is more difficult, because we depend on architects to generate golden sets. Specifically in the case of co-change clusters, we have the impression that expert developers are not completely aware of the whole set of changes implemented in their systems and on the co-change relations established due to such changes, making it more challenging to build a golden set of co-change clusters.

VII. Threats to Validity

First, we evaluated six systems, implemented in two languages (Java and Pharo) and related to two major domains (information systems and software tools). Therefore, our results may not generalize to other systems, languages, and application domains (external validity). Second, our results may reflect personal opinions of the interviewed developers on software architecture and development (conclusion validity).

Anyway, we interviewed expert developers, with large experience, and who are responsible for the central architectural decisions in their systems. Third, our results are directly impacted by the thresholds settings used in the study (internal validity). We handled this threat by reusing thresholds from our previous work on Co-Change Clustering, which were defined after extensive experimental testings. Furthermore, thresholds selection is usually a concern in any technique for detecting patterns in source code. Finally, there are threats concerning the way we measured the co-change relations (construct validity). Specifically, we depend on pre and pos-processing filters to handle commits that could pollute the co-change graphs with meaningless relations. For example, during the analysis with developers, we detected three co-change clusters (6%) that are motivated by small refactorings. Ideally, it would be interesting to discard such commits automatically. Finally, we only measured co-change relations for classes. However, SysPol has other artifacts, such as XML and XHTML files, which are not considered in the study.

VIII. Related Work

Semantic Clustering is a technique based on Latent Semantic Indexing (LSI) and clustering to group source code artifacts that use similar vocabulary [6], [7]. Therefore, Co-Change and Semantic Clustering are conceptually similar techniques, sharing the same goals. However, they use different data sources (commits vs vocabularies) and processing algorithms (Chamaleon graph clustering algorithm vs LSI). Moreover, Semantic Clustering was not evaluated in the field, using developers' views on the extracted clusters.

Ball et al. introduced the concept of co-change graphs [11] and Beyer and Noack improved this concept proposing a graph visualization technique to reveal clusters of frequently co-changed artifacts [12]. Their approach clusters all software artifacts which are represented as vertices in the co-change graphs. In contrast, we prune several classes during the co-change graph and co-change cluster extraction phases. We also compute co-change clusters using a clustering algorithm designed for sparse graphs, as is typically the case of co-change graphs. Finally, our focus is not on software visualization but on using co-change clusters to support modularity analysis.

Co-Change mining is used to predict changes [13], [32], to support program visualization [12], [33], to reveal logical dependencies [34], [35], to improve defect prediction techniques [36], and to detect bad smells [37]. Zimmermann et al. propose an approach that uses association rules mining on version histories to suggest possible future changes (e.g., if class `A` usually co-changes with `B`, and a commit only changes `A`, a warning is raised recommending to check whether `B` should not be changed too) [13]. However, co-change clusters are coarse-grained structures, when compared to the set of classes in association rules. They usually have more classes (at least, four classes according to our thresholds) and are detected less frequently. Therefore, they are better instruments to help developers on program comprehension.

Bavota et al. investigate how the various types of coupling aligns with developer's perceptions [38]. They consider coupling measures based on structural, dynamic, semantic, and logical information and evaluate how developers rate the identified coupling links. Their results suggest that semantic coupling seems to better reflect the developers' mental model that represents interactions between entities. However, the developers interviewed in the study evaluated only pairs of classes. In contrast, we retrieve co-change clusters having four or more classes.

Vanya et al. use co-change clusters to support systems partitioning, aiming to reduce coupling between parts of a system [39]. Their approach differs from ours because co-change clusters containing files from the same part of the system are discarded. Beck and Diehl compare and combine logical and structural dependencies to retrieve modular designs [40], [41]. They report clustering experiments to recover the architecture of ten Java programs. Their results indicate that logical dependencies are interesting when substantial evolutionary data is available. In this paper, we considered all commits available for the evaluated systems (almost 24K commits).

IX. Conclusion

One of the benefits of modularity is managerial, by allowing separate groups to work on each module with little need for communication [1]. However, this is only possible if modules confine changes; crosscutting changes hamper modularity by making it more difficult to assign work units to specific teams. In this paper, we evaluate in the field a technique called Co-Change Clustering, proposed to assess modularity using logical (or evolutionary) coupling, as captured by co-change relations. We concluded that Encapsulated Clusters are very often linked to healthy designs and that 67% of Crosscutting Clusters are associated to design anomalies. Octopus Clusters are normally associated to expected class distributions, which are not easy to implement in an encapsulated way, according to the interviewed developers.

As future work, we plan to conduct a large scale study to analyze whether the occurrence of co-change patterns is associated to programming languages. Particularly, we intend to consider well-known projects implemented in other languages, such as C/C++. Also, we plan to investigate the effect that alternative software architecture styles, e.g., architectures based on domain-driven principles, have on co-change patterns. Specifically, we plan to check whether such architectures can avoid the generation of clusters matching Crosscutting or Octopus patterns. We also plan to compare and contrast the results of Co-Change Clustering with the ones generated by Semantic Clustering [5], [6].

Acknowledgments

This research is supported by grants from FAPEMIG, CAPES, and CNPq. We would like to thank the developers of SysPol and the developers of the evaluated Pharo systems for accepting to participate in our study.

REFERENCES

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules." *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[2] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[3] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[4] N. Anquetil, C. Fourrier, and T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in *6th Working Conference on Reverse Engineering (WCRE)*, 1999, pp. 235–255.

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

[6] A. Kuhn, S. Ducasse, and T. Girba, "Enriching reverse engineering with semantic clustering," in *12th Working Conference on Reverse Engineering (WCRE)*, 2005, pp. 133–142.

[7] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[8] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. de Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 1, pp. 1–33, 2014.

[9] L. L. Silva, M. T. Valente, and M. Maia, "Assessing modularity using co-change clusters," in *13th International Conference on Modularity*, 2014, pp. 49–60.

[10] ——, "Co-change clusters: Extraction and application on assessing software modularity," *Transactions on Aspect-Oriented Software Development (TAOSD)*, pp. 1–37, 2015.

[11] T. Ball, J. K. A. A. Porter, and H. P. Siy, "If your version control system could talk ..." in *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.

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

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

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

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

[16] G. Karypis, E.-H. S. Han, and V. Kumar, "Chameleon: hierarchical clustering using dynamic modeling," *Computer*, vol. 32, no. 8, pp. 68–75, 1999.

[17] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance (ICSM)*, 2004, pp. 350–359.

[18] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[19] O. Nierstrasz, S. Ducasse, and D. Pollet, *Pharo by Example*. Square Bracket Associates, 2010.

[20] L. L. Silva, D. Felix, M. T. Valente, and M. Maia, "ModularityCheck: A tool for assessing modularity using co-change clusters," in *5th Brazilian Conference on Software: Theory and Practice*, 2014, pp. 1–8.

[21] U. Flick, *An Introduction to Qualitative Research*. SAGE, 2009.

[22] J. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *11th European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 1241. Springer Verlag, 1997, pp. 220–242.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *15th European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 2072. Springer Verlag, 2001, pp. 327–355.

[25] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N degrees of separation: Multi-dimensional separation of concerns," in *21st International Conference on Software Engineering (ICSE)*, 1999, pp. 107–119.

[26] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Prentice Hall, 2003.

[27] S. Apel and C. Kästner, "Virtual separation of concerns - A second chance for preprocessors," *Journal of Object Technology*, vol. 8, no. 6, pp. 59–78, 2009.

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

[29] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *31st International Conference on Software Engineering (ICSE)*, 2009, pp. 287–297.

[30] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *IEEE International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[31] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2013, pp. 132–146.

[32] M. P. Robillard and B. Dagenais, "Recommending change clusters to support software investigation: An empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 3, pp. 143–164, 2010.

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

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

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

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

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

[38] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 692–701.

[39] A. Vanya, L. Hofland, S. Klusener, P. van de Laar, and H. van Vliet, "Assessing software archives with evolutionary clusters," in *16th IEEE International Conference on Program Comprehension (ICPC)*, 2008, pp. 192–201.

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

[41] ——, "On the impact of software evolution on software clustering," *Empirical Software Engineering*, vol. 18, no. 5, pp. 970–1004, 2013.