

# AGRUPAMENTO DE CO-MUDANÇAS



LUCIANA LOURDES SILVA

## AGRUPAMENTO DE CO-MUDANÇAS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

ORIENTADOR: MARCO TÚLIO DE OLIVEIRA VALENTE

COORIENTADOR: MARCELO DE ALMEIDA MAIA

Belo Horizonte

30 de outubro de 2015



LUCIANA LOURDES SILVA

## CO-CHANGE CLUSTERING

Thesis presented to the Graduate Program  
in Computer Science of the Federal Univer-  
sity of Minas Gerais in partial fulfillment of  
the requirements for the degree of Doctor  
in Computer Science.

ADVISOR: MARCO TÚLIO DE OLIVEIRA VALENTE

CO-ADVISOR: MARCELO DE ALMEIDA MAIA

Belo Horizonte

October 30, 2015

© 2015, Luciana Lourdes Silva.  
Todos os direitos reservados.

**Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG**

Silva, Luciana Lourdes

S586c Co-Change Clustering / Luciana Lourdes Silva. —  
Belo Horizonte, 2015  
xxvi, 136 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas  
Gerais— Departamento de Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente  
Coorientador: Marcelo de Almeida Maia

1. Computação - Teses. 2. Engenharia de Software -  
Teses. 3. Programação modular - Teses. I. Orientador.  
II. Coorientador. III. Título.

CDU 519.6\*32(043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


## FOLHA DE APROVAÇÃO


Co-change clustering

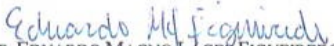
**LUCIANA LOURDES SILVA**

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

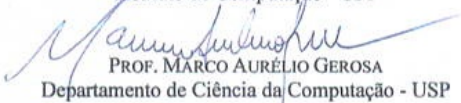
  
PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador  
Departamento de Ciência da Computação - UFMG

  
PROF. MARCELO DE ALMEIDA MAIA - Coorientador  
Departamento de Ciência da Computação - UFU

  
PROF. HUMBERTO TORRES MARQUES NETO  
Instituto de Ciências Exatas e Informática - PUC/MG

  
PROF. EDUARDO MAGNO LAGES FIGUEIREDO  
Departamento de Ciência da Computação - UFMG

  
PROF. LEONARDO CRESTA PAULINO MURTA  
Instituto de Computação - UFF

  
PROF. MARCO AURÉLIO GEROSA  
Departamento de Ciência da Computação - USP

Belo Horizonte, 23 de novembro de 2015.





*This thesis is dedicated to my fiancé Jânio and my mom Zilda, who have always supported me.*

*And also to my honorary father Edson and aunt Rosalina who have strongly influenced on my academic progress.*



# Acknowledgments

I would like to express my gratitude to everyone who supported me throughout my work on this PhD thesis. This work would have been impossible without their support, encouragement, and guidance. I thank God for putting all those people on my path.

First of all, I thank my dear fiancé Jânio for his comprehension, support, and presence when I needed him most. I also thank my mother Zilda who has encouraged me to carry on and tolerated my absence from family.

I would like to express my sincere gratitude to my advisor M. T. Valente for his attention, motivation, patience, dedication, and immense knowledge. I certainly could not complete my Ph.D study without him. In addition, my sincere thanks to my co-advisor M. Maia for your knowledge in datamining and statistics techniques, as well as for continuously encouragement on these years. I would also like to thank my co-advisor N. Anquetil for the essential help to validate this thesis.

I thank my dear friend N. Félix for the several lessons on datamining techniques which were crucial to conduct my work. I would like to thank G. Santos and M. Dias for their great support to validate my work while I was in France. I also thank J. Rosklin for helping me to define the clustering measure used in this thesis. Thank you all my friends G. Cabral, E. Miguel, K. Souza, H. Silva, C. Silva, S. Ribas, R. E. Moreira, and S. M. Dias for your support and help in so many situations, specially during my qualification exams.

I also thank my lovely friends P. Oliveira, B. Vercosa, D. Perdigão, and Q. Damasceno for their constant presence and moral support when I was facing difficult times.



*“The secret of freedom lies in educating people, whereas the secret of tyranny is in keeping them ignorant.”*

(Maximilien de Robespierre)



# Resumo

Modularidade é um conceito chave em projeto de sistemas de *software* complexos. No entanto, decomposição modular ainda é um desafio após décadas de pesquisas para prover novas formas de modularização de sistemas de *software*. Uma razão é que modularização pode não ser vista com uma única perspectiva devido às múltiplas facetas que um *software* tem que lidar. Pesquisas em linguagens de programação ainda tentam definir novos mecanismos de modularização que sejam capazes de lidar com essas diferentes facetas, tal como aspectos e *features*. Adicionalmente, a estrutura modular tradicional definida pela hierarquia de pacotes sofre do problema de decomposição dominante e sabe-se que alternativas de modularização são necessárias para aumentar a produtividade dos desenvolvedores. Para contribuir com uma solução para esse problema, nesta tese nós propusemos um visão modular alternativa para compreender e avaliar modularidade em pacotes baseada em *co-change clusters*, que são arquivos de código inter-relacionados considerando relações de *co-change*. Os *co-change clusters* são classificados em seis padrões considerando suas projeções sobre a estrutura de pacotes: *Encapsulated*, *Well-Confined*, *Crosscutting*, *Black-Sheep*, *Octopus* e *Squid*. A abordagem foi avaliada em três estágios diferentes: (i) uma análise quantitativa em três sistemas do mundo real, (ii) uma análise qualitativa em seis sistemas, implementados em duas linguagens, para revelar a percepção dos desenvolvedores em relação aos *co-change clusters*, (iii) um estudo em larga escala em uma base de 133 projetos hospedados no GitHub implementados em seis linguagens de programação. Por meio dessas análises pode-se concluir que *Encapsulated Clusters* são geralmente vistos como bons módulos e *Crosscutting Clusters* tendem a ter associação com anomalias de projeto quando eles representam interesses entrelaçados. *Octopus Clusters* tem uma associação estatística significativa com efeito em cascata (*ripple effect*), que de acordo com desenvolvedores, não são fáceis de implementar de maneira encapsulada.





# Abstract

Modularity is a key concept to embrace when designing complex software systems. Nonetheless, modular decomposition is still a challenge after decades of research on new techniques for software modularization. One reason is that modularization might not be viewed with single lens due to the multiple facets that a software must deal with. Research in programming languages still tries to define new modularization mechanisms to deal with these different facets, such as aspects and features. Additionally, the traditional modular structure defined by the package hierarchy suffers from the dominant decomposition problem and it is widely accepted that alternative forms of modularization are necessary to increase developer's productivity. In order to contribute with a solution to this problem, in this thesis we propose a novel technique to assess package modularity based on co-change clusters, which are highly inter-related source code files considering co-change relations. The co-change clusters are classified in six patterns regarding their projection to the package structure: Encapsulated, Well-Confined, Crosscutting, Black-Sheep, Octopus, and Squid. We evaluated our technique in three different fronts: (i) a quantitative analysis on four real-world systems, (ii) a qualitative analysis on six systems implemented in two languages to reveal developer's perception of co-change clusters, (iii) a large scale study in a corpus of 133 GitHub projects implemented in six programming languages. We concluded that Encapsulated Clusters are often viewed as healthy designs and that Crosscutting Clusters tend to be associated to design anomalies when they represent tangled concerns. Octopus Clusters have a significant statistical association with ripple effect but they are normally associated to expected class distributions, which are not easy to implement in an encapsulated way.



# List of Figures

1.1	Co-change cluster extraction and visualization . . . . .	5
1.2	Co-change pattern examples. . . . .	6
2.1	Euclidean distance based clustering in a 3-D space. . . . .	13
2.2	Chameleon phases [Karypis et al., 1999]. . . . .	14
2.3	Tokenization example from Lucene’s issues. . . . .	18
2.4	A stop word list. . . . .	18
2.5	Ten issue reports (documents) after applying the text pre-processing tasks.	19
2.6	An example of term-document matrices. . . . .	19
2.7	The LSA space. . . . .	20
2.8	The LSA space in text format matrix. . . . .	20
2.9	Document-document matrix after applying cosine similarity. . . . .	21
2.10	An Example of a Distribution Map [Ducasse et al., 2006]. . . . .	21
2.11	Co-change graph example. . . . .	29
3.1	Multiple commits for the issue GERONIMO-3003 . . . . .	41
3.2	Single commit handling multiple issues (3254, 3394 to 3398) . . . . .	41
3.3	Highly scattered commit (251 changed classes) . . . . .	42
3.4	Packages changed by commits in the Lucene system . . . . .	42
3.5	Co-change clusters size (in number of classes) . . . . .	49
3.6	Co-change clusters density . . . . .	50
3.7	Cluster average edges’ weight . . . . .	50
3.8	Focus . . . . .	51
3.9	Spread . . . . .	51
3.10	Focus versus Spread . . . . .	52
3.11	Distribution map for Geronimo . . . . .	53
3.12	Part of the Distribution map for Lucene . . . . .	55
3.13	Part of the Distribution map for JDT Core . . . . .	57

3.14	Distribution map for Camel . . . . .	58
3.15	Examples of heat maps for similarity of issues . . . . .	61
3.16	Distribution of the clusters' score . . . . .	62
4.1	Encapsulated clusters (Glamour) . . . . .	70
4.2	Well-Confined cluster (Intellij-Community) . . . . .	70
4.3	Crosscutting cluster (SysPol) . . . . .	71
4.4	Black Sheep cluster (Ruby) . . . . .	72
4.5	Octopus cluster (Moose) . . . . .	72
4.6	Squid cluster (Platform Frameworks) . . . . .	73
4.7	ModularityCheck's overview. . . . .	74
4.8	ModularityCheck's architecture. . . . .	75
4.9	Filters and metric results. . . . .	76
5.1	Co-change cluster sizes . . . . .	82
5.2	Distribution map metrics . . . . .	82
5.3	Distribution map metrics . . . . .	83
5.4	Octopus cluster . . . . .	88
6.1	The overall number of commits and number of files by language . . . . .	97
6.2	Commit reduction rate for 5, 10, and 15 minutes . . . . .	99
6.3	Evolution of commit density (source code files) per co-change pattern . . . . .	100
6.4	MoJoFM values for 600 clusterings . . . . .	102
6.5	Relative number of identified and classified clusters for each system . . . . .	103
6.6	Relative Co-Change Pattern coverage by Programming Language . . . . .	104
6.7	Effects on Co-change Bursts . . . . .	107
6.8	Effects on Activity of Clusters. E - Encapsulated, C - Crosscutting, O - Octopus, and NP - Clusters with no Pattern . . . . .	108
6.9	Effects on Activity of Clusters. E - Encapsulated, C - Crosscutting, O - Octopus, and NP - Clusters with no Pattern . . . . .	109
6.10	Effects on Number of Developers of Clusters. E - Encapsulated, C - Crosscutting, O - Octopus, and NP - No Pattern . . . . .	110
6.11	Effects on Commits of Owners. E - Encapsulated, C - Crosscutting, O - Octopus, and NP - Clusters with no Pattern . . . . .	112

# List of Tables

2.1	Examples of spread and focus calculus. . . . .	23
2.2	Summary of empirical studies on recommendation and modularity analysis. . . . .	37
3.1	Target systems (size metrics) . . . . .	45
3.2	Initial commits sample . . . . .	45
3.3	Percentage of unitary commits (i.e., changing a single class) discarded in the first phase and commits discarded after each preprocessing filters . . . . .	47
3.4	Number of vertices ( $ V $ ), edges ( $ E $ ) and co-change graphs' density (D) before and after the post-processing filter . . . . .	47
3.5	Number of co-change clusters . . . . .	48
3.6	Maintenance issues in Cluster 28 . . . . .	56
3.7	Correlation between score, focus and spread of clusters for Geronimo, Lucene, JDT Core, and Camel . . . . .	63
5.1	Target systems . . . . .	79
5.2	Preprocessing filters and Number of co-change clusters . . . . .	80
5.3	Number of vertices ( $ V $ ), edges ( $ E $ ), co-change graphs' density (D), and number of connected components before and after the post-processing filter . . . . .	80
5.4	Number of co-change clusters . . . . .	81
5.5	Expert Developers' Profile . . . . .	84
5.6	Number and percentage of co-change patterns . . . . .	85
5.7	Concerns implemented by encapsulated clusters . . . . .	85
5.8	Concerns implemented by Crosscutting clusters in SysPol . . . . .	86
5.9	Concerns implemented by Octopus clusters . . . . .	88
6.1	Projects in GitHub organized by language . . . . .	98
6.2	Number and percentage of categorized co-change clusters . . . . .	102
6.3	Number and percentage of categorized co-change clusters grouped in three major patterns . . . . .	103

6.4	NBR model for number of co-change bursts per system. . . . .	106
6.5	Deviance table for NBR model on the number of co-change bursts per system. . . . .	107
6.6	NBR model for number of commits per cluster. C - Crosscutting Clusters, NP - Clusters with no Pattern, O - Octopus Clusters . . . . .	107
6.7	Deviance table for NBR model on the number of commits per cluster . . .	109
6.8	NBR model for number of developers per cluster . . . . .	110
6.9	Deviance table for NBR model on the number of developers per cluster . .	111
6.10	NBR model for commit number of cluster's owner . . . . .	111
6.11	Deviance table for NBR model on the commit number of cluster's owner .	112
6.12	The projects with the most massively changed clusters . . . . .	113
6.13	Ranking of Encapsulated Clusters . . . . .	114
6.14	Ranking of Crosscutting Clusters . . . . .	115
6.15	Ranking of Octopus Clusters . . . . .	116
6.16	Timeline for the top one Encapsulated Cluster . . . . .	116
6.17	Timeline for the top one Crosscutting Cluster . . . . .	117
6.18	Timeline for the top one Octopus Cluster . . . . .	118

# Contents

<b>Acknowledgments</b>	<b>xi</b>
<b>Resumo</b>	<b>xv</b>
<b>Abstract</b>	<b>xvii</b>
<b>List of Figures</b>	<b>xix</b>
<b>List of Tables</b>	<b>xxi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem . . . . .	1
1.2 Research Contributions . . . . .	4
1.3 Publications . . . . .	8
1.4 Thesis Outline . . . . .	8
<b>2 Background</b>	<b>11</b>
2.1 Clustering Techniques . . . . .	11
2.1.1 Data and Clustering Concepts . . . . .	11
2.1.2 Chameleon: A Graph Clustering Algorithm . . . . .	13
2.1.3 Agglomerative Merging Function . . . . .	15
2.2 Latent Similarity Analysis . . . . .	17
2.2.1 Text Pre-processing Tasks . . . . .	18
2.2.2 Applying LSA and Pre-processing Tasks . . . . .	18
2.2.3 Cosine Similarity . . . . .	20
2.3 Distribution Map . . . . .	20
2.4 Version History Analysis . . . . .	23
2.5 Software Modularity Views . . . . .	25
2.5.1 Traditional Package Decomposition . . . . .	26

2.5.2	Textual Approaches . . . . .	26
2.5.3	Concern Approaches . . . . .	27
2.5.4	Co-change Approaches . . . . .	28
2.5.5	Hybrid Approaches . . . . .	34
2.5.6	Critical Analysis . . . . .	36
2.6	Final Remarks . . . . .	37
<b>3</b>	<b>Co-Change Clustering</b>	<b>39</b>
3.1	Proposed Technique . . . . .	39
3.1.1	Extracting Co-Change Graphs . . . . .	39
3.1.2	Extracting Co-Change Clusters . . . . .	43
3.2	Co-Change Clustering Results . . . . .	44
3.2.1	Target Systems and Thresholds Selection . . . . .	45
3.2.2	Co-Change Graph Extraction . . . . .	46
3.2.3	Co-Change Clustering . . . . .	48
3.3	Modularity Analysis . . . . .	50
3.3.1	Distribution Map for Geronimo . . . . .	52
3.3.2	Distribution Map for Lucene . . . . .	54
3.3.3	Distribution Map for JDT Core . . . . .	56
3.3.4	Distribution Map for Camel . . . . .	57
3.4	Semantic Similarity Analysis . . . . .	59
3.4.1	Pre-processing Issue Description . . . . .	59
3.4.2	Latent Semantic Analysis . . . . .	60
3.4.3	Scoring clusters . . . . .	60
3.4.4	Correlating Similarity, Focus, and Spread . . . . .	62
3.5	Discussion . . . . .	63
3.5.1	Practical Implications . . . . .	63
3.5.2	Clustering vs Association Rules Mining . . . . .	64
3.6	Threats to Validity . . . . .	65
3.7	Final Remarks . . . . .	66
<b>4</b>	<b>Co-Change Patterns</b>	<b>69</b>
4.1	Proposed Co-Change Patterns . . . . .	69
4.2	ModularityCheck . . . . .	73
4.2.1	ModularityCheck in a Nutshell . . . . .	74
4.2.2	Architecture . . . . .	74
4.3	Final Remarks . . . . .	76



<b>5</b>	<b>Developers' Perception on Co-change Clusters</b>	<b>77</b>
5.1	Study Design . . . . .	77
5.1.1	Research Questions . . . . .	77
5.1.2	Target Systems and Developers . . . . .	78
5.1.3	Thresholds Selection . . . . .	78
5.1.4	Extracting the Co-Change Clusters . . . . .	80
5.1.5	Interviews . . . . .	83
5.2	Results . . . . .	84
5.2.1	To what extent do the proposed co-change patterns cover real instances of co-change clusters? . . . . .	84
5.2.2	How developers describe the clusters matching the proposed co-change patterns? . . . . .	85
5.2.3	To what extent do clusters matching the proposed co-change patterns indicate design anomalies? . . . . .	88
5.3	Discussion . . . . .	90
5.3.1	Applications on Assessing Modularity . . . . .	90
5.3.2	The Tyranny of the Static Decomposition . . . . .	91
5.3.3	(Semi-)Automatic Remodularizations . . . . .	91
5.3.4	Limitations . . . . .	92
5.4	Threats to Validity . . . . .	92
5.5	Final Remarks . . . . .	93
<b>6</b>	<b>A Large Scale Study on Co-Change Patterns</b>	<b>95</b>
6.1	Study Design . . . . .	95
6.1.1	Research Questions . . . . .	95
6.1.2	Dataset . . . . .	96
6.1.3	Threshold Selection . . . . .	98
6.2	Clustering Quality . . . . .	99
6.2.1	Analysis of the Co-Change Cluster Extraction . . . . .	100
6.3	Classifying Co-Change Clusters . . . . .	101
6.4	Statistical Methods . . . . .	104
6.5	Analysis of the Results . . . . .	105
6.5.1	Statistical Results . . . . .	105
6.5.2	Semantic Analysis . . . . .	112
6.6	Threats to Validity . . . . .	119
6.7	Final Remarks . . . . .	119

<b>7 Conclusion</b>	<b>121</b>
7.1 Summary . . . . .	121
7.2 Contributions . . . . .	122
7.3 Further Work . . . . .	123
<b>Bibliography</b>	<b>125</b>

# Chapter 1

## Introduction

In this chapter, we first introduce the main problem investigated in this thesis (Section 1.1). Next, we present our objectives, contributions (Section 1.2), and an overview of our technique for assessing modularity of software systems using co-change dependencies (Section 1.2). Section 1.3 presents the publications derived from this thesis. Finally, we present the thesis outline (Section 1.4).

### 1.1 Problem

Software systems must continuously support changes to requirements, otherwise they become progressively useless [Lehman, 1984; Parnas, 1994]. Nonetheless, the original program modularization may degrade over time, then, reducing its quality [Bavota et al., 2014]. Thus, the software complexity tends to increase unless attempts to limit the degradation effect of the changes are performed [Lehman, 1984]. If the complexity is mismanaged, software maintenance become increasingly hard [Sarkar et al., 2009] and developers may need to spend significant effort to comprehend and apply changes. Consequently, maintenance activities can become extremely expensive [Sarkar et al., 2009]. It is estimated that change costs tend to exceed 70 percent of the software total cost [Meyer, 2000; Seacord et al., 2003].

In recent years much attention has been focused on reducing software cost [Zimmermann et al., 2005; Abdeen et al., 2011; Bavota et al., 2014; Palomba et al., 2013]. A software implementation that reduce the effort to comprehend and apply changes is crucial for reducing maintenance costs. Particularly, software complexity is an important factor that affects program comprehension [Curtis et al., 1979]. However, comprehensibility can improve depending on how the system is designed. The common strategy to reduce complexity is to obtain a modular design. Modularity is a key concept to

embrace when designing complex software systems [Baldwin and Clark, 2003]. The central idea is to organize the system into manageable parts—named modules—that can be easily understood.

Dijkstra [1974] introduced the notion of separation of concerns, which is an essential principle in software development and architecture design. More specifically, this principle advocates that one module should represent a single concern. Separation of concerns can be achieved through modular design that involves the software decomposition in encapsulated units [Laplante, 2003]. However, some concerns cannot be easily encapsulated and, consequently, their scattering over the system’s modules may be a signal of modularity flaws [Kiczales et al., 1997a; Walker et al., 2012]. For instance, bad design decisions on separating concerns can increase the amount of structural dependencies between the modules of a system (high coupling).

In his seminal paper on modularity and information hiding, Parnas [1972] developed the principle that modules should hide *“difficult design decisions or design decisions which are likely to change”*. Similarly, DeRemer and Kron [1975] defined that, in a modular design, the modules must implement distinct parts of the system and work together as a whole. The modules should be designed and then implemented independently from each other. Furthermore, modularity should accommodate future uncertainty because the specific elements in a modular design may change in latter stages [Baldwin and Clark, 2006]. In addition, according to Meyer [2000], modularity covers the combination of extensibility and reusability, which are major factors to assess external software quality. Consequently, the maintenance and evolution tasks become easier because the modules can be understood and changed individually [Parnas, 1972; DeRemer and Kron, 1975]. For instance, eventually a system evolves, new modules may be replaced for older ones at low cost and in an effortless way. The recommended practice to decompose a system in packages keeps the classes that have to change together as close as possible. Therefore, it is fundamental to consider modularity when assessing the internal quality of software systems [Martin, 2003; Meyer, 2000; Kiczales et al., 1997a].

Parnas’ principle has deeply impacted the way that systems were designed and implemented in the last forty years. Nowadays, any developer with a minimal training on software engineering try to follow Parnas’ insights, consciously or by using programming languages and tools that foster information hiding principles. However, the criteria proposed by Parnas to decompose systems into modules are not widely used to assess whether — after years of maintenance and evolution — the modules of a system are 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 from a structural perspective, using static measures of size, coupling, cohesion, etc [Mitchell and Mancoridis, 2006; Chidamber and Kemerer, 1994; Anquetil et al., 1999; Chidamber and Kemerer, 1991; Stevens et al., 1974].

Specifically, the standard approach to assess modularity relies on structural dependencies established between the modules of a system (coupling) and between the internal elements from each module (cohesion). Usually, high cohesive and low-coupled modules are desirable because they ease software comprehension, maintenance, and reuse. However, structural cohesion and coupling metrics measure a single dimension of the software implementation (the static-structural dimension). Moreover, it is widely accepted that traditional modular structures and metrics suffer from the dominant decomposition problem and tend to hinder different facets that developers may be interested in [Kersten and Murphy, 2006; Robillard and Murphy, 2002, 2007]. The tyranny of the dominant decomposition refers to the problem of decomposing a system into modular units. Typically, there are concerns—most of them are non-functional ones—that crosscut modules on the system, reducing software quality in terms of comprehension and evolution [Mens et al., 2004].

In order to mitigate the limitation of structural dependencies, several attempts have been made for assessing software modularity. Diverse approaches rely on different dimensions, such as use cases [Ratiu et al., 2009], dynamic information [Ostermann et al., 2005], software evolution [Mileva and Zeller, 2011; D’Ambros et al., 2009a], and semantic relations—normally extracted from source code vocabularies using information retrieval algorithms [Santos et al., 2014; Kuhn et al., 2005, 2007]. Less frequently, some works combine static information with evolutionary [Beck and Diehl, 2010] and semantic relations [Kagdi et al., 2013; Bavota et al., 2014, 2013]. These distinct modularity views can detect effects of coupling not captured by structural metrics. In the context of the dynamic view, call relationships between classes that occur during program execution are captured to measure software coupling [Arisholm et al., 2004]. In contrast, under an evolutionary dimension, logical coupling can be extracted from sets of files that often change together without being structurally close to each other [Schwanke et al., 2013]. Therefore, to improve current modularity views, it is important to investigate the impact of design decisions concerning modularity in other dimensions of a software system, as the evolutionary one.

Despite the large effort in research aiming to define an effective technique for assessing software modularity, there is no solution widely accepted. This thesis aims to address the logical dimension by offering an in-depth investigation on the capability of the modules in a software system to confine changes after years of maintenance and evolution. Typically, the evolutionary approaches are centered on association rules to

detect co-change relations [Zimmermann et al., 2005]. Nonetheless, they do not consider the projection of code file sets that usually change together in the traditional decomposition of a system in modules, such as packages. Our hypothesis for assessing the degree of modularity follows Parnas' principle, i.e., the greater the number of changes localized in modules, the better the modularity. However, some changes propagate to other modules but they are associated to expected class distributions due to the complexity to implement in an encapsulated way. Therefore, we propose a modularity assessment technique that allows developers to investigate how often changes are localized in modules and to check whether crosscutting changes reveal design problems.

## 1.2 Research Contributions

Since the first attempts to understand the benefits of modular programming, the relevance of evolutionary aspects are highlighted. Despite that, we still lack widely accepted and tested techniques to assess modularity under an evolutionary perspective. Therefore, the main goal of this PhD work is to propose a new technique to assess modularity centered on sets of software artifacts that usually changed together in the past, which we term *co-change clusters* [Silva et al., 2014b, 2015a]. The proposed technique also classifies co-change clusters in recurrent patterns regarding their projection to the package structure.

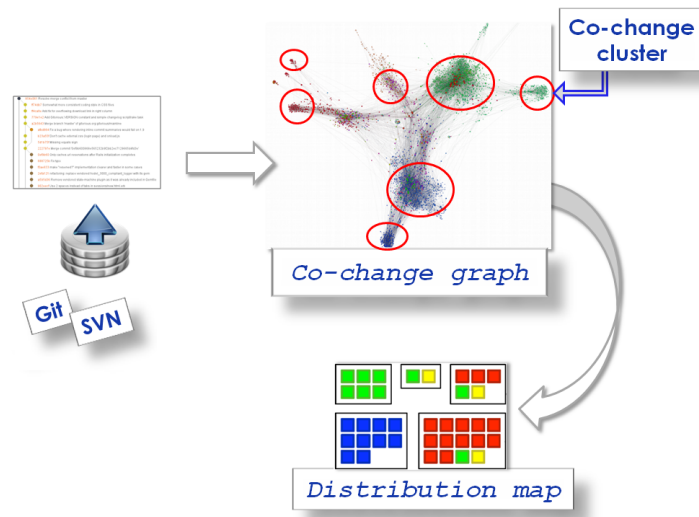
The contributions proposed in this thesis can be organized in seven specific ones that together achieved the overall goal of our work, as follows:

1. We propose and evaluate of a technique based on logical dependencies for assessing modularity at the class level. Particularly, we contrast the co-change modularity with the standard package decomposition. The technique and results are described in Chapter 3.
2. We define recurrent patterns of co-change clusters denoting Encapsulated, Well-confined, Crosscutting, Black-sheep, Octopus, and Squid Clusters. These co-change patterns are presented in Chapter 4.1.
3. We design and implement a prototype tool for the visual exploration and analysis of co-change cluster patterns, named ModularityCheck [Silva et al., 2014c]. This tool is presented in Chapter 4.2.
4. We collect the perception of expert developers on co-change clusters, aiming to answer two central questions: (i) what concerns and changes are captured by the

extracted clusters? (ii) do the extracted clusters reveal design anomalies? ([Silva et al., 2015b]) This study design, results, and lessons learned are described in Chapter 5.

5. We conduct a series of empirical analysis in a large corpus of the most popular software projects in GitHub, aiming 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? The dataset, study design, quantitative and qualitative results are described in Chapter 6.

The technique proposed in this thesis is directly inspired by the common criteria used to decompose systems in modules, i.e., modules should confine implementation decisions that are likely to change [Parnas, 1972]. Figure 1.1 summarizes our technique. We first extract co-change graphs from the change history in software systems [Beyer and Noack, 2005]. In such graphs, the nodes are classes and the edges link classes that were modified together in the same commit transaction. After that, co-change graphs are automatically processed to produce a new modular facet: co-change clusters, which abstract out common changes made to a system, as stored in version control platforms. Thus, co-change clusters represent sets of classes that frequently changed together in the past.

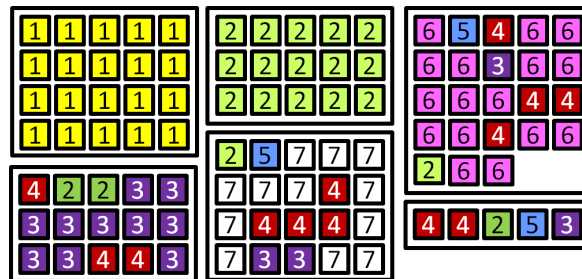


**Figure 1.1.** Co-change cluster extraction and visualization

Furthermore, our technique relies on distribution maps [Ducasse et al., 2006] — a well-known visualization technique (Figure 1.1)—to reason about the projection

of the extracted clusters in the traditional decomposition of a system in packages. We then rely on a set of metrics defined for distribution maps to characterize the extracted co-change clusters. We use six recurrent distribution patterns of co-change clusters (five patterns were borrowed from distribution map technique). Our technique allows developers to visualize distribution maps for the system (clusters projected over packages) and investigate whether the co-change relations represent the expected class distribution. The patterns guide developers to localize change propagation between packages and analyze whether such changes represent design problems, e.g., a concern is not well encapsulated or a class is in the wrong package. Figure 1.2 depicts examples of projected co-change clusters on distribution maps.

- Encapsulated Clusters (Cluster 1) - clusters that when projected over the package structure match all co-change classes of their packages.
- Well-Confined Clusters (Clusters 6 and 7) - clusters whose classes are confined in a single package.
- Crosscutting Clusters (Cluster 4) - clusters whose classes are spread over several packages, touching few classes in each one.
- Black-Sheep Clusters (Cluster 5) - similar behavior to Crosscutting, but they touch fewer classes and packages.
- Octopus Clusters (Cluster 2) - clusters that have most classes in one package and some “tentacles” in other packages.
- Squid Clusters (Cluster 3) - similar to Octopus, the difference is in the body’s size, which is smaller than Octopus’ body.



**Figure 1.2.** Co-change pattern examples.

Three patterns—Encapsulated, Crosscutting, and Octopus Clusters—emerged after an investigation to assess the modularity of four real-world systems [Silva et al.,



2015a] and the others, after our qualitative study on six systems [Silva et al., 2015b]. For instance, in our first study, most co-change clusters in two systems are Encapsulated, suggesting that the system’s package are well-modularized. In counterpart, the co-change clusters in another system contains types of patterns that may suggest modularity flaws: several co-change clusters present a Crosscutting behavior and some follow the Octopus pattern, which may indicate a possible ripple effect during maintenance activities. Moreover, we also evaluated the semantics of the obtained clusters using information retrieval techniques. The goal in this particular case was to understand how similar the issues whose commits were clustered together are.

To reveal developers’ view on the usage of Co-Change Clustering, we conducted an empirical study with experts on six systems. One system is a closed-source and large information system implemented in Java and six systems are open-source software tools implemented in Pharo (a Smalltalk-like language). Our goal in this analysis is to investigate the developer’s perception of co-change clusters for assessing modular decompositions. We mined 102 co-change clusters from the version histories of such systems, which were then classified in three patterns regarding their projection over the package structure. 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 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 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—reveal design (or modularity) flaws.

To conclude our work, we conducted a series of empirical analysis in a large corpus of 133 popular projects hosted in GitHub implemented in six languages. In this last study, we considered all six patterns observed in our previous works. We mined 1,802 co-change clusters from the version histories of such projects, which were then categorized in six patterns regarding their projection over the package structure. From the initially computed clusters, 1,719 co-change clusters (95%) are covered by the proposed co-change patterns. Our goal in this final study is to evaluate whether the occurrence of certain co-change patterns are associated to programming language and how different co-change patterns relate to ripple effects, level of activity on clusters, the number of developers working on the clusters, and the level of ownership on clusters. Finally, we also analyze clusters with high level of activity to understand the rationale behind the considered co-change patterns and how they evolve overtime.

## 1.3 Publications

Much of the work contained in this thesis was published in conferences and journals listed below. Each publication is noted in parenthesis for the corresponding chapters.

1. Silva, L. L., Valente, M. T., and Maia, M. (2014). Assessing modularity using co-change clusters. In 13th International Conference on Modularity, pages 49–60. **Best Paper Award.** (Chapter 3)
2. Silva, L. L., Felix, D., Valente, M. T., and Maia, M. (2014). ModularityCheck: A tool for assessing modularity using co-change clusters. In 5th Brazilian Conference on Software: Theory and Practice, pages 1–8. **3rd Best Tool Award.** (Chapter 4.2)
3. Silva, L. L., Valente, M. T., and Maia, M. (2015). Co-change clusters: Extraction and application on assessing software modularity. In Transactions on Aspect-Oriented Software Development XII, volume 8989 of Lecture Notes in Computer Science, pages 96–131. Springer. (Chapter 3)
4. Silva, L. L., Valente, M. T., Maia, M., and Anquetil, N. (2015). Developers' perception of co-change patterns: An empirical study. In 31st IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 21–30. (Chapters 4.1 and 5)

## 1.4 Thesis Outline

This thesis is structured in the following chapters:

- Chapter 2 provides basic concepts on co-change graphs, hierarchical clustering techniques, Latent Similarity Analysis (LSA), software modularity, and distribution maps. This chapter also presents the state-of-the-art on techniques of assessing modularity.
- Chapter 3 describes the modularity assessment technique proposed in this work, as well its evaluation.
- Chapter 4 presents six co-change pattern definitions evaluated in this thesis. This chapter also presents the ModularityCheck tool for the visual exploration and analysis of co-change clusters.

- Chapter 5 presents a qualitative evaluation of our technique with experts on six systems, implemented in two languages (Pharo and Java).
- Chapter 6 presents a large scale study using projects hosted in GitHub with a series of empirical analyses.
- Chapter 7 concludes this PhD thesis.



# Chapter 2

## Background

In this chapter, before discussing related work, we present fundamental concepts related to our thesis. We begin by introducing concepts related to clustering techniques and the clustering algorithm used in this work (Section 2.1). Next, we present concepts of information retrieval based on Latent Semantic Analysis (LSA) and linguistic pre-processing, which can be used to reason about the vocabulary of a system (Section 2.2). Furthermore, we describe a visualization technique—distribution maps—used in this work (Section 2.3). Additionally, we review works related to version history analysis (Section 2.4). Then, we present an overview on different modularity views related to our work (Section 2.5). Finally, we conclude with a general discussion (Section 2.6).

### 2.1 Clustering Techniques

This section provides a background on the clustering techniques. We present the traditional techniques available, such as hierarchical and partitioning clustering. We use clustering techniques to retrieve clusters of software artifacts that tend to change together—co-change clusters.

#### 2.1.1 Data and Clustering Concepts

Clustering techniques aim to support the automated and non-trivial extraction of implicit, unknown and potentially useful information from large databases [Tan et al., 2006]. In general, data mining tasks can be classified as predictive and descriptive. The former constructs models, with the goal to predict the behavior of unknown data sets; whereas the later finds interpretable patterns and relationships that describe the underlying data. Unlike predictive task, a descriptive task does not predict new proper-

ties but it guides the exploration of properties of the target data set. Particularly, this section covers the Clustering concepts, a specific descriptive task, since our technique currently plays upon clustering analysis extracted from co-change graphs.

The understanding of clustering approaches is relevant to our study, since our approach relies on graph clustering to find co-change clusters. In this section, we present data preprocessing and partitioning and hierarchical clustering techniques (Section 2.1), a graph clustering algorithm called Chameleon (Section 2.1.2), and agglomerative merging schemes used to adapt Chameleon to our purpose (Section 2.1.3).

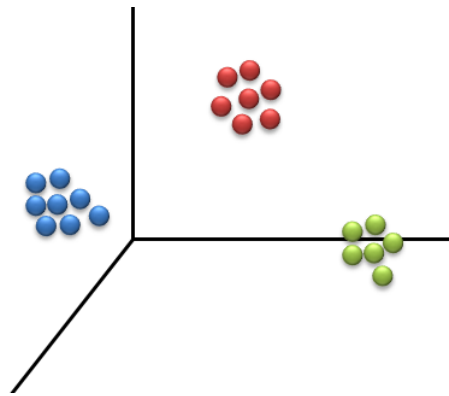
Knowledge mining from data can be described as a process that consists of several steps. Below we present the major steps in data preprocessing:

- Data cleaning — remove noise and outlier data. There are data objects that are referred as outliers and noise [Han et al., 2011]. A data object is considered an outlier when it deviates significantly from the normal objects. It violates the source that generates the normal data objects. Some application examples: credit card fraud detection and medical analysis. Outliers are different from noisy data, in which a measured variable contains random errors or variance, i.e., incorrect attribute values such as faulty data collection instruments, duplicated records, and incomplete or inconsistent data.
- Data integration — merge multiple data sources.
- Data reduction — retrieve important data from the database, such as dimensionality reduction by removing unimportant attributes.
- Data transformation — apply functions that map the entire set of values of given attribute to a new set, such as normalization techniques.
- Data mining — apply techniques to extract data patterns, such as clustering approaches.

Our technique combines data cleaning, data reduction, and data mining steps. Firstly, we eliminate outliers and noisy data. Then, we reduce the dimensionality by discarding infrequent attributes. Finally, we use a clustering algorithm to retrieve co-change clusters.

Cluster analysis [Tan et al., 2006] is the process of grouping data objects into subsets, where a subset is a collection of objects that are similar to each other. Clustering techniques, as illustrated in Figure 2.1, should produce high quality clusters by ensuring that intra-cluster distances are minimized (high similarity) and inter-cluster distances

are maximized (low similarity). Commonly, distance functions, such as the Euclidean Distance function, and similarity functions are used to compute similarity. To support software analysis and maintenance activities, several clustering algorithms have been used to automatically partition a system into semantic clusters [Wu et al., 2005]. Software clustering has been applied to decompose systems into meaningful modules, usually to support system re-modularization [Vanya et al., 2008], comprehension [Kothari et al., 2006; Robillard and Dagenais, 2010], and architecture recovery [Anquetil and Lethbridge, 1999; Maqbool and Babri, 2007].

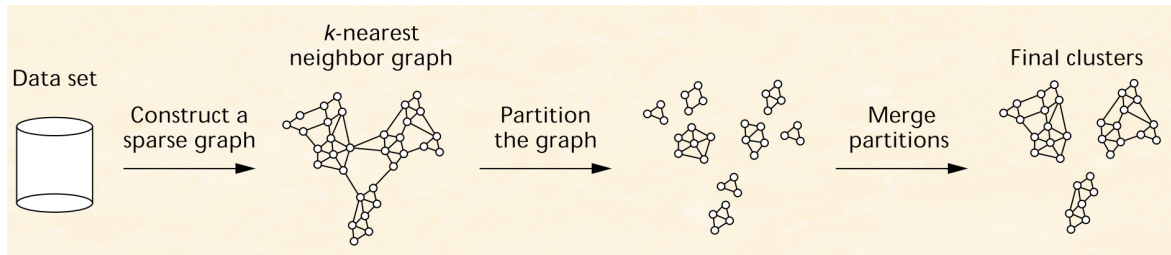


**Figure 2.1.** Euclidean distance based clustering in a 3-D space.

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

### 2.1.2 Chameleon: A Graph Clustering Algorithm

Chameleon [Karypis et al., 1999] is an agglomerative hierarchical clustering algorithm designed to handle sparse graphs in which nodes represents data object, and weighted edges represent similarities among the data objects. Figure 2.2 provides an overview of the approach, which is divided into the steps described as follows:



**Figure 2.2.** Chameleon phases [Karypis et al., 1999].

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

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

The clustering algorithm consists of two-phases: i) Partition the Graph and ii) Merge Partitions.

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

**Second Phase - Merge Partitions:** This phase uses an agglomerative hierarchical algorithm to merge the small clusters, created by the first phase, repeatedly. Clusters are combined to maximize the number of links within a cluster (internal similarity) and to minimize the number of links between clusters (external similarity). Chameleon models similarity based on the Relative Interconnectivity (RI) and Relative Closeness (RC) of the clusters. A pair of clusters  $C_i$  and  $C_j$  are selected to merge when both RI and RC are high, suggesting that they are well interconnected as well as close together.  $RI(C_i, C_j)$  is their absolute interconnectivity normalized regarding their



internal interconnectivity. To retrieve the internal interconnectivity of a cluster, the algorithm sums the edges crossing a min-cut bisection that splits the cluster into two equal parts. The absolute closeness of clusters is the average weight of the edges that connect vertices between  $C_i$  and  $C_j$ . To retrieve the internal closeness, the algorithm takes the average of the edge weights across a min-cut bisection that splits the cluster into two equal parts. Similarly to RI,  $RC(C_i, C_j)$  is the absolute closeness normalized regarding the internal closeness of two clusters  $C_i$  and  $C_j$ .

**User-specified Thresholds.** In this step, those pairs of clusters in which the user-specified thresholds exceed relative interconnectivity ( $T_{RI}$ ) and relative closeness ( $T_{RC}$ ) are merged. For each cluster  $C_i$ , Chameleon searches for clusters  $C_j$  that exceed RI and RC thresholds.

### 2.1.3 Agglomerative Merging Function

Chameleon’s authors implemented a software package, named Cluto<sup>1</sup>, which allows to use different agglomerative merging schemes in the Chameleon’s second phase. As our goal is to find co-change clusters in a sparse graph, Chameleon is an appropriated algorithm to cluster data objects — in our case, software artifacts — in a co-change graph because they are robust concerning to noisy data. The software artifacts discarded by Chameleon are considered noisy data by the algorithm because they do not belong to any cluster. The Cluto package includes both traditional and novel agglomerative merging schemes as follows [Manning et al., 2008; Sneath and Sokal, 1973]:

*SLINK - Single-link hierarchical clustering.* This scheme merges in each step the two clusters which have the smallest distance. In other words, the distance between clusters  $C_i$  and  $C_j$  is the *minimum* distance between any data object in  $C_i$  and any data object in  $C_j$ . The SLINK scheme is recommended to handle non-globular shapes, but it is sensitive to noise and outliers producing long clusters. Mahmoud and Niu [2013] proposed an approach which uses change transactions (commits) to evaluate clustering algorithms in the context of program comprehension. They analyzed partitioning, hierarchical agglomerative, and comprehension-based clustering algorithms. Their results showed that hierarchical algorithms are the most suitable in recovering comprehension decompositions, with the exception of the Single-link scheme. Moreover, instability analysis showed that hierarchical algorithms are more stable than other clustering algorithms.

---

<sup>1</sup><http://glaros.dtc.umn.edu/gkhome/views/cluto>.

*CLINK - Complete-link hierarchical clustering.* This scheme merges in each step the two clusters whose union has the smallest diameter. In other words, the distance between clusters is defined by the two most distant objects in the distinct clusters. CLINK yields balanced clusters, with equal diameter and it is less susceptible to noise. However, it usually breaks large clusters because all clusters tend to have an equal diameter, and the small clusters are merged with the larger ones.

*UPGMA - Unweighted Pair Group Method with Arithmetic Mean.* This scheme first identifies among all subclusters the two clusters that are more similar to each other and then consider them as a single cluster, which is referred as a composite cluster. Subsequently, among the new group of clusters it repeatedly identifies the pair with the highest similarity, until we reach only two clusters. This is the default merging scheme in the Cluto package.

*I1 - Cohesion based on graphs.* This scheme maximizes the sum of the average pairwise similarities between the objects assigned to each cluster, weighted according the size of the clusters, as follows:

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

where

$k$  = number of subclusters defined in the first phase.

$n_i$  = number of objects in the  $i$ -th cluster.

$S_i$  = set of objects assigned to the  $i$ -th cluster.

$\text{sim}(v, u)$  is the similarity between vertices  $v$  and  $u$ .

*I2 - Cohesion based on graphs.* This function searches for subclusters to combine, maximizing the similarity by evaluating how close are the objects in a cluster, as follows:

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

After an exhaustive experiment, we selected the scheme *I2* because it represents better our goals. In other words, it increases cluster densities while maximizing the internal similarity.

## 2.2 Latent Similarity Analysis

This section provides a background on information retrieval techniques. We present the traditional techniques available, such as linguistic preprocessing and LSA. We use linguistic preprocessing, LSA, and cosine similarity techniques to evaluate the meaning of issue reports related to software maintenance activities.

The discussion of Latent Similarity Analysis (LSA) [Deerwester et al., 1990] is relevant to our technique, since our work evaluates the semantic similarity of issue reports that are related to a specific cluster in order to improve our understanding of the clusters' meaning. LSA is a statistical approach for extracting and representing the meaning of words. The semantic information is retrieved from a word-document co-occurrence matrix, where words and documents are considered as points in an Euclidean space. LSA is based on the Vector Space Model (VSM), an algebraic representation of documents frequently used in information retrieval [Salton et al., 1975]. The vector space of a text collection is constructed by representing each document as a vector with the frequencies of its words. The document vectors add to a term-by-document matrix representing the full text collection. First, the vocabulary of terms is determined using feature selection techniques such as tokenization, stop words removal, domain vocabulary, case-folding, stemming and weighting schemes (TF-IDF, binary weight) before representing the text data in a numerical form. Moreover, LSA applies singular value decomposition (SVD) to the term-by-document matrix as a way of factor analysis. Singular value decomposition is performed on the matrix to determine patterns in the relationships between the terms and concepts contained in the text. In SVD, a rectangular matrix is decomposed into the product of three other matrices—an orthogonal matrix  $U$ , a diagonal matrix  $\Sigma$ , and the transpose of an orthogonal matrix  $V$ . Suppose an original term-document matrix  $C_{M \times N}$ , where  $M$  is the number of terms and  $N$  is the number of documents. The matrix  $C$  is then decomposed via SVD into the term vector matrix  $U$ , the document vector matrix  $V$ , and the diagonal matrix  $\Sigma$  (consisting of eigenvalues) as follows:

$$C_{M \times N} = U_{M \times M} \Sigma_{M \times N} V_{N \times N}^T$$

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

In Subsection 2.2.2 we present a detailed example of how LSA works on documents.

## 2.2.1 Text Pre-processing Tasks

When analyzing text documents, an adequate pre-processing step is crucial to achieve good results [Manning et al., 2008]. After collecting the documents to be analyzed, some steps are usually performed as follows:

**Tokenization.** The tokenization process is applied on the text chopping character streams into tokens, discarding special characters, such as punctuation and numbers. Furthermore, in software artifacts, the CamelCase identifiers are also split into tokens. Figure 2.3 shows an example of tokenization.

Input: exceptions, being, swallowed, include a patch that logs the problem in the catch clause

Output: 

exceptions	being	swallowed	include	a	patch	that	logs	the	problem	in	the	catch	clause
------------	-------	-----------	---------	---	-------	------	------	-----	---------	----	-----	-------	--------

**Figure 2.3.** Tokenization example from Lucene's issues.

**Stop words removal.** In this step, common words which are irrelevant when selecting documents matching an end-user needs are removed from the vocabulary. Figure 2.4 shows an example of a stop word list.

a	an	and	are	as	at	be	by	for	from
has	he	in	is	it	its	on	that	the	to
was	were	will	with						

**Figure 2.4.** A stop word list.

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

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

## 2.2.2 Applying LSA and Pre-processing Tasks

In order to illustrate how LSA and the aforementioned pre-processing tasks work, we selected a small sample of documents, as presented in Figure 2.5. The terms in the figure were stemmed, the stopwords were dropped, and the letters were reduced to lower case. Figure 2.6 shows the term-by-document matrices with binary frequency (1 if the term is present in the document and 0 otherwise). The matrix in the left side is

```

[1] facet lucen taxonomy writer reader
[2] lucen index facet taxonomy
[3] lucen index
[4] facet java lucen search taxonomy reader searcher manag
[5] facet java apach lucen taxonomy directory
[6] facet java lucen children array parent
[7] core lucen index commit
[8] facet lucen index
[9] facet lucen taxonomy
[10] facet lucen taxonomy directory array parallel reader

```

**Figure 2.5.** Ten issue reports (documents) after applying the text pre-processing tasks.

Terms	Docs									
	1	2	3	4	5	6	7	8	9	10
apach	0	0	0	0	1	0	0	0	0	0
array	0	0	0	0	0	1	0	0	0	1
children	0	0	0	0	0	1	0	0	0	0
commit	0	0	0	0	0	0	1	0	0	0
core	0	0	0	0	0	0	1	0	0	0
directory	0	0	0	0	1	0	0	0	0	1
facet	1	1	0	1	1	1	0	1	1	1
index	0	1	1	0	0	0	1	1	0	0
java	0	0	0	1	1	1	0	0	0	0
lucen	1	1	1	1	1	1	1	1	1	1
manag	0	0	0	1	0	0	0	0	0	0
parallel	0	0	0	0	0	0	0	0	0	1
parent	0	0	0	0	0	1	0	0	0	0
reader	1	0	0	1	0	0	0	0	0	1
search	0	0	0	1	0	0	0	0	0	0
searcher	0	0	0	1	0	0	0	0	0	0
taxonomy	1	1	0	1	1	0	0	0	1	1
writer	1	0	0	0	0	0	0	0	0	0

Terms	Docs									
	1	2	3	4	5	6	7	8	9	10
array	0	0	0	0	0	1	0	0	0	1
directory	0	0	0	0	1	0	0	0	0	1
facet	1	1	0	1	1	1	0	1	1	1
index	0	1	1	0	0	0	1	1	0	0
java	0	0	0	1	1	1	0	0	0	0
lucen	1	1	1	1	1	1	1	1	1	1
reader	1	0	0	1	0	0	0	0	0	1
taxonomy	1	1	0	1	1	0	0	0	1	1

**Figure 2.6.** An example of term-document matrices.

the term-document matrix representing the documents in Figure 2.5. In the right side, we have a matrix after pruning terms which appeared only once in all documents. The goal is to select important words to a issue report in a collection.

Figure 2.7 shows the LSA space of the Figure 2.6 after pruning non frequent terms. The matrix  $tk$  represents the term vector matrix  $U$ , the matrix  $dk$  represents the document vector matrix  $V$ , and  $sk$  represents the diagonal matrix  $\Sigma$ . The resulting latent semantic space can be converted back to text format matrix, as shown in Figure 2.8. Rows are terms, columns are documents. This allows to make comparisons between terms and documents.

```

$tk
      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]
array  -0.1378128 -0.16932952 -0.42425535  0.67518466 -0.09430368  0.28373790 -0.25522131 -0.40240517
directory -0.1550758 -0.23081805 -0.09368788  0.31898385  0.76946818 -0.31338425  0.28422025  0.20266787
facet  -0.5373984 -0.09126795 -0.05810750 -0.15046650 -0.07407766  0.59241189  0.55429826  0.11476305
index  -0.1779987  0.76551538  0.04554012  0.11699877  0.07742641 -0.20116222  0.31293447 -0.47132895
java   -0.2091017 -0.22708104 -0.64075838 -0.46553776 -0.12965946 -0.43533438  0.06767414 -0.25788873
lucen  -0.5973299  0.33393756 -0.11052597  0.06984138 -0.08392599 -0.15059571 -0.47285609  0.51114910
reader -0.2270878 -0.30795804  0.37920673  0.35064235 -0.53602039 -0.46547036  0.28765719  0.01086147
taxonomy -0.4293986 -0.25649299  0.48907521 -0.24797744  0.27635153  0.03173775 -0.37146609 -0.48350420

$dk
      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]
1  -0.3521415 -0.141500210  0.4443218  0.01692281 -0.363546105  0.008307344 -0.004241277  0.34193873
2  -0.3424909  0.330549155  0.2324221 -0.16247567  0.170403792  0.279932648  0.041056776 -0.73381126
3  -0.1524247  0.483473600 -0.0412702  0.14346142 -0.005657295 -0.361496069 -0.286586976  0.08883735
4  -0.3932496 -0.241356859  0.0373990 -0.34053092 -0.476402921 -0.439078920  0.117033930 -0.23340201
5  -0.3790925 -0.207435307 -0.2629195 -0.36483925 -0.659906677 -0.282782424  0.110874765  0.19451135
6  -0.2912816 -0.067606067 -0.7834454  0.09906676 -0.332467507  0.298254190 -0.190145086 -0.07670448
7  -0.1524247  0.483473600 -0.0412702  0.14346142 -0.005657295 -0.361496069 -0.286586976  0.08883735
8  -0.2580738  0.443339420 -0.0781722  0.02792877 -0.070135193  0.247316265  0.706741260  0.34486971
9  -0.3074975 -0.006078694  0.2035011 -0.25231080  0.103011113  0.486663883 -0.519736335  0.31770716
10 -0.4097215 -0.317461164  0.1153945  0.78027493  0.224123458 -0.022159747  0.047726173 -0.10366821

$sk
[1] 5.0866336 2.2740703 1.5746434 1.3023721 1.1488846 0.9730616 0.5580212 0.4482365

```

Figure 2.7. The LSA space.

	1	2	3	4	5	6	7	8	9	10
array	-1.249	5.551	1.110	-2.983	-3.677	1.000	6.938	4.857	6.938	1.000
directory	-6.071	-9.714	-3.400	-2.255	1.000	-8.239	-4.232	-6.730	-3.087	1.000
facet	1.000	1.000	-5.698	1.000	1.000	1.000	-6.097	1.000	1.000	1.000
index	-1.942	1.000	1.000	-2.220	3.469	9.714	1.000	1.000	-1.249	-4.267
java	-7.563	-2.775	-4.145	1.000	1.000	1.000	-3.989	-8.951	-1.734	-1.196
lucen	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
reader	1.000	-7.181	-4.866	1.000	-8.789	-3.829	-4.088	-4.527	-4.267	1.000
taxonomy	1.000	1.000	2.428	1.000	1.000	-1.838	-5.551	-4.718	1.000	1.000

Figure 2.8. The LSA space in text format matrix.

### 2.2.3 Cosine Similarity

Cosine similarity measures the cosine of the angle between two vectors  $\vec{v}_i$  and  $\vec{v}_j$  corresponding to the documents  $d_i$  and  $d_j$  in the semantic space constructed by LSA,  $sim(\vec{v}_i, \vec{v}_j) \in [-1, 1]$ :

$$sim(\vec{v}_i, \vec{v}_j) = \frac{\vec{v}_i \bullet \vec{v}_j}{|\vec{v}_i| \times |\vec{v}_j|}$$

where  $|\vec{v}|$  is the vector norm and  $\bullet$  is the vector internal product operator.

Figure 2.9 shows the document-document matrix obtained from LSA space depicted in Figure 2.7.

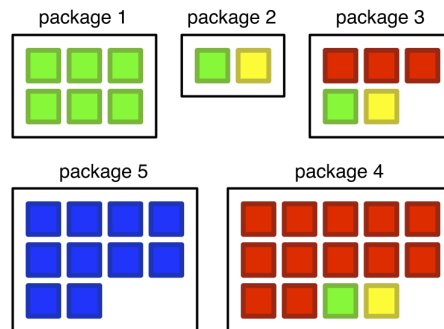
## 2.3 Distribution Map

Distribution map [Ducasse et al., 2006] has been used in analysis of bugs [Hora et al., 2012] and cluster semantic analysis [Kuhn et al., 2007; Santos et al., 2014]. It is a generic technique to make easier the visualization and analysis of how a property

	1	2	3	4	5	6	7	8	9	10
1	1.0000000	0.7500000	0.3535534	0.8944272	0.6708204	0.5000000	0.3535534	0.5773503	0.8660254	0.8164966
2	0.7500000	1.0000000	0.7071068	0.6708204	0.6708204	0.5000000	0.7071068	0.8660254	0.8660254	0.6123724
3	0.3535534	0.7071068	1.0000000	0.3162278	0.3162278	0.3535534	1.0000000	0.8164966	0.4082483	0.2886751
4	0.8944272	0.6708204	0.3162278	1.0000000	0.8000000	0.6708204	0.3162278	0.5163978	0.7745967	0.7302967
5	0.6708204	0.6708204	0.3162278	0.8000000	1.0000000	0.6708204	0.3162278	0.5163978	0.7745967	0.7302967
6	0.5000000	0.5000000	0.3535534	0.6708204	0.6708204	1.0000000	0.3535534	0.5773503	0.5773503	0.6123724
7	0.3535534	0.7071068	1.0000000	0.3162278	0.3162278	0.3535534	1.0000000	0.8164966	0.4082483	0.2886751
8	0.5773503	0.8660254	0.8164966	0.5163978	0.5163978	0.5773503	0.8164966	1.0000000	0.6666667	0.4714045
9	0.8660254	0.8660254	0.4082483	0.7745967	0.7745967	0.5773503	0.4082483	0.6666667	1.0000000	0.7071068
10	0.8164966	0.6123724	0.2886751	0.7302967	0.7302967	0.6123724	0.2886751	0.4714045	0.7071068	1.0000000

**Figure 2.9.** Document-document matrix after applying cosine similarity.

crosscuts several parts of the system or whether it is well-encapsulated. Distribution maps are typically used to compare two partitions  $P$  and  $Q$  of the entities from a system  $S$ . Moreover, entities are represented as small squares and the partition  $P$  groups such squares into large rectangles. Finally, partition  $Q$  is used to color the squares. Figure 2.10 shows an example of a distribution map containing 5 packages, 37 classes, and 4 semantic clusters.



**Figure 2.10.** An Example of a Distribution Map [Ducasse et al., 2006].

**Reference partition.** The partition  $P$  refers to a well defined partition. It represents the intrinsic structure of the system, e.g., the package structure or the result of a data clustering. Large squares in Figure 2.10 represent reference partition.

**Comparison partition.** The partition  $Q$  is the result of a specific analysis. Mostly, it refers to a set of clusters or mutually exclusive properties. To distinguish the comparison partition from the reference partition, we refer to the former using the term *cluster*. The colors in Figure 2.10 represent comparison partition.

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

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

where

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

In this definition,  $touch(q, p_i)$  is the number of elements of cluster  $q$  located in the partition  $p_i$  divided by the number of elements in  $p_i$  that are included in at least a cluster. Similarly,  $touch(p_i, q)$  is the number of elements in  $p_i$  included in the cluster  $q$  divided by the number of elements in  $q$ . Focus ranges between 0 and 1, where the value one means that the cluster  $q$  dominates the parts that it touches, i.e., it is well-encapsulated in such parts.

Furthermore, distribution maps can be used to quantify the *spread* of a given cluster  $q$  in relation to the partition  $P$ , as follows:

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

Ducasse et al. [2006] proposed a vocabulary with various recurring patterns, but we present below only patterns which aim at characterizing co-change clusters retrieved in this work.

**Well-encapsulated.** This kind of cluster is similar to the reference partition. In our case, a reference partition represents a package structure. The cluster can be spread over one or multiple packages, but it should include almost all elements within those modules. In Figure 2.10, the cluster blue is well-encapsulated, since it dominates the package 5.

**Cross-Cutting.** This kind of cluster is spread over several partitions, but only touches them superficially. In Figure 2.10, the cluster yellow cross-cuts packages 2, 3, and 4.

**Octopus.** This kind of cluster dominates one package, but also spreads across other packages, like cross-cutting cluster. In Figure 2.10, the cluster green covers the package 1 and also spread over packages 2, 3, and 4.



**Black Sheep.** This kind of cluster crosscuts the system, but it touches very few elements.

Table 2.1 shows examples of spread and focus extracted from [Ducasse et al., 2006]. These values were obtained from Figure 2.10.

**Table 2.1.** Examples of spread and focus calculus.

Color	Size	Spread	Focus	Description
red	15	2	0.80	main property
blue	11	1	1.0	well-encapsulated
green	9	4	0.75	octopus
yellow	3	3	0.25	crosscutting

## 2.4 Version History Analysis

In this section, we review some works on version history analysis that reveal some concerns they should be taken into account in this kind of data analysis (set of software artifacts that usually change together). A recent study by Negara et al. [2012] reveals that the use of data from version history present many threats when investigating source code properties. For example, developers usually fix failing tests by changing the test themselves, commit tasks without testing, commit the same code fragment multiple times (in different commits), or take days to commit changes containing several types of tasks.

Kawrykow and Robillard [2011] developed a tool for detecting non-essential differences in change sets extracted from version histories. Non-essential differences are defined as low-level code changes that are cosmetic and usually preserve the behavior of the code. For example, a non-essential modification can be local a rename refactoring, where all methods that contain references to the renamed element will also be textually changed. They claimed that such non-essential code modifications can lead approaches, that are based on historical data, to induce inaccurate high-level interpretations of software development effort. Their detection technique works on a level of granularity finer than statement differences because non-essential differences occur within statements or expressions. The results showed that between 2.6% and 15.5% of all method updates were non-essential modifications. To analyze whether non-essential differences would interfere in the result obtained by change-based approaches, they implemented a method to mine association rules similar to the one proposed by Zimmermann et al. [2005]. They evaluated the quality of the recommendations produced when all method

updates were used to retrieve rules against their quality considering only essential updates. Their approach improved the overall precision of the recommendations by 10.5% and decreased their recall by 4.2%.

Tao et al. [2012] conducted a large-scale study at Microsoft to investigate the role of understanding code change processes. The quantitative result was obtained from an online survey, determining that understanding code changes is a frequent practice in major development phases. The qualitative result was obtained from a series of follow-up email interviews, investigating how to improve the effectiveness and efficiency of the practice in understanding code changes. For example, they concluded that it is difficult to acquire important information needs to assess quality of a change such as completeness and consistency because it lacks tool support. Moreover, they investigated the effect of tangled changes (they refer to it as “composite changes”) in the context of change understanding. For understanding a tangled change, it is fundamental to decompose it into sub-changes aligned with their respective issue report. They provided evidence that understanding such changes requires non-trivial efforts and a tool support for change decomposition.

Herzig and Zeller [2013] investigated five open-source Java programs and manually classified more than 7,000 change sets as being tangled code changes (group of distinct changes implemented in a single commit). Such tangled changes can lead approaches based on version history to consider all changes to all modules as being related, compromising the resulting analysis. The result of the manual classification shows that 15% of all bug fixing change sets are tangled changes and these can introduce noise in change data sets. They also proposed a multi-predictor untangling algorithm and showed that on average, at least 16.6% of all source files are incorrectly associated with bug reports. Dias et al. [2015] improved the algorithm proposed by Herzig and Zeller [2011] to untangle changes that were submitted in the same commit transaction. Their approach relies on three machine learning algorithms to train models that would predict whether two changes should be in the same cluster. Then, the train models that consist of pair of changes are clustered to be presented to the user. Their results reached 88% of accuracy in determining whether two changes should belong to the same cluster. Finally, they also evaluated the approach by deploying it to 7 developers during 2 weeks and automatically created clusters of untangled changes with a median success rate of 91% and average of 75%.

In this work, we propose pre-processing and post-processing tasks to tackle some threats concerning the use of data from version history. For instance, to discard commits associated to non-essential changes we can remove commits that modify a massive number of classes [Walker et al., 2012]. As another example, if the project provides bug

reports we can ease the problem concerning fragmented maintenance tasks in multiple commits and tangled changes. For the first problem, we can merge commits related to the same maintenance Issue-ID and the second we can discard commits associated to multiple maintenance issues.

## 2.5 Software Modularity Views

In this section, we briefly present some relevant concepts of modular software design, since our goal in this thesis relies on assessing modularity using co-change clusters. Further, we discuss the state-of-the-art on different modularity views, such as traditional package decomposition, concern-driven, co-change, and hybrid approaches.

According to Parnas [1972], a module represents a responsibility assignment. During software development, changes are performed constantly in tasks related to new features, code refactoring, and bug fixing. In a modular software, when one of these tasks needs to be made, it should change a single module with minimal impact in other modules [Aggarwal and Singh, 2005]. The benefits of a modular software are flexibility improvement, comprehensibility, and reduction of system development time [Parnas, 1972]. These benefits are described as follows:

- **Flexibility.** Well-designed modules enables drastic changes in one module with no need to change others.
- **Comprehensibility.** Assuming that modules represent concerns, the comprehension of the system can be achieved by studying a module at a time.
- **Managerial.** Well-designed modules should be as independent as possible. Therefore, the development time shortened by allowing distinct groups to work in different modules independently.

A module has high cohesion if it contains concerns related to each other and keeps out other concerns. High-cohesive modules make the system as a whole easier to understand and to make changes. On the other hand, two modules with high coupling occurs when there are interdependencies between them. As a result, a change in one module can require changes in other modules. Besides, the strong interdependencies among modules makes it hard to understand at a glance how modules work.

### 2.5.1 Traditional Package Decomposition

Abdeen et al. tackled the problem of assessing modularity in large legacy object-oriented systems [Abdeen et al., 2011]. They provided a set of coupling and cohesion metrics to assess packages organization related to package changeability, reusability, and encapsulation. The metrics were defined considering the types of inter-class dependencies, method call and inheritance relationships. They presented a theoretical validation of their proposed coupling and cohesion metrics by demonstrating that their metrics satisfy the mathematical properties.

Wang et al. proposed two quality metrics (succinctness and coverage) to measure the quality of mined usage patterns of API methods [Wang et al., 2013]. They also proposed UP-Miner, a tool to mine succinct and high-coverage API usage patterns from source code. They evaluated their approach quantitatively on a Microsoft code base and compared the results with those obtained from MAPO [Zhong et al., 2009]. Moreover, they also conducted a qualitative study with Microsoft developers and observed that UP-Miner is more effective and outperforms MAPO.

### 2.5.2 Textual Approaches

Maletic and Marcus [2000] used LSA to retrieve relevant clusters of files using only textual analysis. The clusters were used to assist in the comprehension of complex systems. They evaluated a system with 95 KLOC with no external documentation available and analyzed how the clusters help in comprehension and maintenance activities. Kuhn et al. [2007, 2005] have proposed an approach to retrieve the linguistic topics in the source code vocabulary to help program comprehension. The authors introduced Semantic Clustering—a technique based on Latent Semantic Index (LSI)—and clustering to group software artifacts with similar vocabulary. These clusters denote linguistic topics, such as identifier names and comments, that reveal the intention of the code. They applied LSI again to label automatically each cluster with its most important terms to represent it. Finally, they used the distribution maps to visualize the spread of the semantic clusters over the system.

Recently, Santos et al. [2014] proposed an approach based on Semantic Clustering to evaluate software re-modularization. They adapted semantic clustering to help in software re-modularization. The process to extract semantic clusters differs from Kuhn et al. [2007] on stop criteria. Kuhn et al. fixed a number of nine clusters independently of the system size. On the other hand, Santos et al. [2014] replaced the stop criteria by a threshold, where the agglomerative hierarchical clustering algorithm merges clusters until all pairs have similarity lower than the threshold value. Moreover, their approach

relies on a set of metrics that measure conceptual cohesion, spread, and focus. Finally, they used distribution maps to visualize how the semantic clusters are distributed over the system.

Semantic Clustering is a technique based on Latent Semantic Analysis (LSA) and clustering to group source code artifacts that use similar vocabulary. 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.

### 2.5.3 Concern Approaches

Several approaches were proposed to help developers and maintainers to manage concerns and features. For example, concern graphs model the subset of a software system associated with a specific concern [Robillard and Murphy, 2002, 2007]. The main purpose is to provide developers with an abstract view of the program fragments related to a concern. FEAT is a tool that supports the concern graph approach by enabling developers to build concern graphs interactively, as result of program investigation tasks. Aspect Browser [Griswold et al., 2001] and JQuery [Janzen and Volder, 2003] are other tools that rely on lexical or logic queries to find and document code fragments related to a certain concern. ConcernMapper [Robillard and Weigand-Warr, 2005] is an Eclipse Plug-in to organize and view concerns using a hierarchical structure similar to the package structure. However, in such approaches, the concern model is created manually or based on explicit input information provided by developers. Moreover, the relations between concerns are typically syntactical and structural. On the other hand, in the approach proposed in this thesis, the elements and relationships are obtained by mining the version history. Particularly, relationships express co-changes and concerns are retrieved automatically by extracting co-change clusters. Indeed, we could feed ConcernMapper with co-change concerns defined by the co-change clusters.

Radiu et al. presented a technique based on use-cases to assess logical modularity of programs [Ratiu et al., 2009]. They mapped the domain knowledge to the program modules that implement it. Their approach captures the structural decomposition to represent the program abstraction. They also extracted semantic domain assuming that a program can be divided into categories such as, persistence, user interfaces, and business domain. Finally, they analyzed the similarities between the concepts (e.g., the concepts of the class Dog should embrace the behavior of a dog, nothing else) and program element names. Several heuristics were used to assess the program logical modularization identifying a concern scattered along modules, different concerns

referenced in the same module, and misplacement of classes in packages. In contrast, our technique bases on issue reports and evolutionary information to assess modularity.

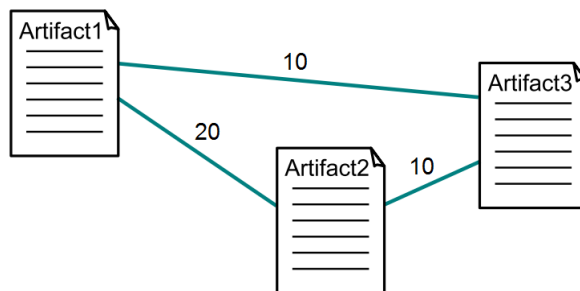
Dit et al. provided a set of benchmarks from Java programs to support evaluation of several software engineering tasks, such as feature location, developer recommendations, and impact analysis [Dit et al., 2013]. The datasets contain textual description of issue reports, method names that were modified when a issue report was implemented, and execution traces. In addition, they introduced the idea of gold set, where for each IssueID, it contains the full qualified method names (i.e., package name, class name, and method name and signature) that were changed when a bug was fixed or when a feature was added. The gold sets contain only SVN commits that explicitly address in their log messages the IssueIDs. Similarly, in this thesis our technique uses issue reports to detect co-change relations that represent maintenance tasks. However, our goal is not provide a set of benchmarks but mining co-change clusters to support on modularity assessment.

## 2.5.4 Co-change Approaches

Co-change mining is used to predict changes [Zimmermann et al., 2005; Robillard and Dagenais, 2010], to support program visualization [Beyer and Noack, 2005; D’Ambros et al., 2009a], to reveal logical dependencies [Alali et al., 2013; Oliva et al., 2011], to improve defect prediction techniques [D’Ambros et al., 2009b], and to detect bad smells [Palomba et al., 2013].

Ball et al. [1997] introduced the concept of co-change graphs and Beyer and Noack [2005] improved this concept and proposed a visualization of such graphs to reveal clusters of frequently co-changed artifacts. A co-change graph is an abstraction for a version control system (VCS). Suppose a set of change transactions (commits) in a VCS, defined as  $T = \{T_1, T_2, \dots, T_n\}$ , where each transaction  $T_i$  changes a set of software artifacts. Conceptually, a co-change graph is an undirected graph  $G = \{V, E\}$ , where  $V$  is a set of artifacts and  $E$  is a set of edges. An edge  $(C_i, C_j)$  is defined between artifacts (vertices)  $C_i$  and  $C_j$  whenever there is a transaction  $T_k$ , such that  $C_i, C_j \in T_k$ , for  $i \neq j$ . Finally, each edge has a weight that represents the number of transactions changing the connected artifacts. Figure 2.11 depicts an example of co-change graph. The edge between *Artifact1* and *Artifact2* denotes that such artifacts frequently change together. In fact, as represented by the edge’s weight, there are 20 transactions that change both artifacts. Their approach clusters all co-change artifacts (source code files, configuration scripts, documentation, etc). The vertex color represents its respective cluster. These vertices are displayed as circles and their area is proportional to the

frequency that the file was changed. In this work we use the concept of co-change graphs presented by Ball et al. [1997] to extract co-change clusters. Their technique differs from ours because we propose preprocessing and post-processing steps, and our focus is not on software visualization but on modularity analysis.



**Figure 2.11.** Co-change graph example.

Oliva et al. mined association rules from version histories to extract logical dependencies between software artifacts for identifying their origins [Oliva et al., 2011]. They conducted a manual investigation of the origins of logical dependencies by reading revision comments and analyzing code diffs. They conclude that the logical dependencies involved files which changed together for different reasons, like Java package renamed, refactoring elements pertaining to a same semantic class, changes in header of Java files, and annotations package created. Their technique differs from ours because we use issue reports and several preprocessing steps to filter out commits that do not represent recurrent maintenance tasks, e.g., package rename and refactoring tasks. Moreover, we mine and classify co-change clusters in recurrent patterns regarding their package decomposition.

Zimmermann et al. [2005] proposed ROSE, an approach that uses association rule mining on version histories to suggest possible future changes and to warn missing changes. Basically, they rely on association rules to recommend further changes, e.g., if class  $A$  usually co-changes with  $B$ , and a commit only changes  $A$ , a warning is given suggesting to check whether  $B$  should be changed too. ROSE pre-processes the system's version history to extract changes at file level from commit transactions and also parse these files to find which entities (functions and fields) were changed. As a final pre-processing step, the large transactions are discarded to remove noise data. In their evaluation, they extracted association rules from the pre-processed data and considered only the top ten single-consequent rules ranked by confidence, assuming that the user does not work with an endless list of suggestions. On average, ROSE can predict 33% of all changed entities. In 70% of all transactions, the three topmost suggestions contain a correct entity. On the other hand, for rapidly evolving systems, the most

useful suggestions of ROSE are at the file level. For example, ROSE predicted 45% of all changed files for the system KOFFICE against 24% for finer granularity. Kagdi and Maletic mined document repositories (web pages) that frequently co-change in a single language or multiple languages [Kagdi and Maletic, 2006]. Their approach searches for documents that co-change in a specific temporal order by mining sequential-patterns. These patterns were used to define groups of entities that need to be changed together in a single or multiple versions. They used the revision numbers to determine the order in which the documents changed, i.e., change-sets with greater revision numbers occurred after those with lower revision numbers. The recovered patterns provided information about the documents that are likely to be re-translated or updated in a single version and in a series of versions. For example, the pattern  $\{kexi.po\} \rightarrow \{krita.po\}$  indicates that these documents were changed in two successive versions in the open-source *KDE* system. This pattern was found in the change history of 506 translated documents across 44 languages, where it can be used to analyze the impact of changes related to translation. However, co-change clusters are coarse-grained structures, when compared to the set of classes in association rules or sequential-patterns. They usually have more classes—at least, four classes according to our thresholds—and are detected less frequently. Moreover, our goal is not recommend changes or analyze change impact concerning translation but to use co-change clusters to evaluate whether classes in terms of co-change are confined in modules.

Wong et al. presented CLIO, an approach that detects and locates modularity violations [Wong et al., 2011]. CLIO compares how components should co-change according to the modular structure and how components usually co-change retrieving information from version history. A *modularity violation* is detected when two components usually change together but they belong to different modules, which are supposed to evolve independently. First, CLIO extracts structural coupling that represents how components should change together. Then, it extracts change coupling that represents how components really change together by mining revision history. Finally, CLIO identifies modularity violations by comparing the results of structural coupling with the results of change coupling. They evaluated CLIO using Hadoop Common and Eclipse JDT. Their results suggested that the detected modularity violations show various signs of poor design, some of which are not detectable through existing approaches. In contrast, our technique relies on logical coupling to extract co-change clusters and then classify them in recurrent patterns regarding their projection to the package structure. Finally, developers can detect modularity violations by analyzing the categorized clusters that reveal scattered changes and ripple effects.

D’Ambros et al. [2009a] presented Evolution Radar, a technique to integrate and



visualize logical coupling information at module and file level, in a single visualization. Their visualization technique shows the dependencies between a module in focus and all the other modules of a system. A file is placed according to the logical coupling that it has with the module in focus. Their tool helps to understand about the evolution of a system (when modules are removed or added), the impact of changes at both file and module levels, and the need for system re-modularization. For example, they were able to detect design issues such as God classes, module dependencies not described in the documentation, and misplaced files. Similarly, our technique also enables developers to locate issues that were detected by Evolution Radar. Their technique differs from ours because some patterns of co-change clusters warn change propagation to support developers on modularity assessment.

Kothari et al. [2006] proposed an approach to characterize the evolution of systems by using change clusters to classify different code change activities (e.g., feature addition, maintenance, etc.) in a time period. First, they identified changes that are as dissimilar as possible and best represent the modifications activities within a period of time. Then, the change clusters were obtained by using clustering techniques, classifying all other changes in a time period as being similar to one of the changes selected previously. The number of changes and the number of change clusters show how much change occurred in a system and the areas where changes have been occurred can be identified, allowing managers see when developers has focused on several or few tasks. In contrast, our goal is not to classify code change activities but allowing developers to comprehend whether changes in the system are often localized in packages or scattered over several packages. For example, developers can inspect clusters which reveal change propagation to analyze whether such changes represent design anomalies.

Alali et al. [2013] provided two measures to improve the accuracy of logical couplings by using frequent pattern mining on version history. The existing approaches suffer from high false-positive rates (i.e., they generate many inaccurate or non relevant claims of coupling). They investigated the problem of reducing the number of false-positive by ranking the mined patterns. It was introduced two measures, namely the age of a pattern and the distance among items within a pattern. The distance describes the position of several files relative to one another within the directory tree. They mined frequent patterns from eleven open-source systems. The results showed that patterns usually denote localized changes (above 75%) with 0 and 2 distances between co-change files. This fact may be an indicative of hidden dependencies and it can help developers in unusual couplings. About the age of a pattern, they observed that there is a chance of 40% that a pattern will occur again within four days. In contrast, we reduce the number of false-positive by considering issue reports to retrieve

commits that represent maintenance activities. Oliva et al. [2013] proposed an approach to assess design decay based on commits in version histories. They focused on the assessment of two particular design smells: rigidity and fragility. The former refers to ripple effect in which changes in a module can propagate to dependent modules; whereas the latter refers to designs with tendency to break the change in different parts every time a change is performed. Similar to our work, their approach discards commits that do not change class files and highly scattered commits. After data collection process, they measured rigidity by calculating the number of changed files per commit and fragility by calculating the distance among file paths contained in a commit. Moreover, we do not consider distance between co-change files as used by Alali et al. [2013] and Oliva et al. [2013]. Instead, we classify co-clusters in recurrent patterns regarding their projection to the package structure (some patterns reveal the presence of ripple effect and scattered changes between co-change classes).

Vanya et al. [2008] used co-change clusters to support the partitioning of system, reducing the coupling between parts of a system. Their approach detects co-change clusters in which a group of files from one part of the system often changes with a group from another part. They applied the technique in a real industrial system containing more than 7 MLOC, evolved during more than a decade. First, change sets that are considered to have a common concern were identified by grouping modifications based on developer and time stamp. After clustering the change sets, they pruned clusters containing files from the same part of the system. The remaining clusters are sorted in decrease order of their cohesion that means how frequently a set of files co-changed. The ordered list of clusters is a suggestion to developers as to which potential evolution issues should be addressed first. Their approach differs from ours because co-change clusters containing files from the same part of the system are not discarded. Moreover, we categorize co-change clusters to warn co-change relations that reveal change propagation.

Robillard and Dagenais [2010] evaluated on seven open-source systems whether change clusters can support developers in their investigation of a software. A change cluster consists of change sets with a certain amount of elements (fields and methods) in common. Their approach discards commit transactions that contain too few or too many changed elements before clustering the transactions. The remaining transactions are clustered by applying a nearest-neighbor clustering algorithm based on the number of overlapping elements. Furthermore, they retrieved clusters that matched to a query and filtered out these clusters by applying four heuristics. A quantitative analysis revealed that less than one in five tasks overlapped with a change cluster. The qualitative analysis of the recommended clusters showed that only 13% of the recommendation for

applicable change tasks were feasible to be useful. Kourosfar [2013] investigated the impact of co-change dispersion on software quality. His results revealed that co-changes localized in the same subsystem involve fewer bugs than co-changes crosscutting the distinct subsystems. In contrast, in this thesis our goal is not to recommend changes or associate scattered changes to bugs but supporting modularity analysis.

Palomba et al. [2013] proposed HIST, a technique that uses association rule mining on version histories to detect the code smells: Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. HIST bases on changes at method level granularity. For each smell, they defined a heuristics to use the association rules discovery or analyze co-changed classes/methods for detecting the distinct bad smells. For example, the association rules between methods of the same class identify classes affected by Divergent Change. They evaluated HIST performances in terms of precision and recall against a manually-produced oracle, and compared with code smell detection techniques based on static source code analysis [Moha et al., 2010; Tsantalis and Chatzigeorgiou, 2009; Fokaefs et al., 2011]. Their technique differs from ours because our goal is not detect code smell but use co-change clusters—coarse grained-structures—to support developers to evaluate the package decomposition of the system.

Breu and Zimmermann [2006] proposed an approach (HAM) that defines their notion of transaction, which is the set of methods inserted by the developer to complete a single development task. They consider that method calls inserted in eight or more locations (method bodies) define aspect candidates. Although this threshold was justified based on their previous experience on finding common error patterns from revision histories, the authors agree that for some projects lower thresholds may be required. The approach ranks the aspect candidates based on three criteria: 1) the size of the set of locations where method calls were inserted, 2) penalization in case of method calls that appears in many transactions, 3) benefit candidates that were introduced in one transaction and extended to additional locations in other transactions. They consider not only methods that were changed together, but also those changes that were the same, i.e., the same method call was inserted. Moreover, this is a more fine-grained notion of change that is interested in finding methods calls to define aspect candidates. One important difference from their work and ours is that they consider not only methods that were changed together, but also those changes that were the same, i.e., the same method call was inserted. Moreover, this is a more fine-grained notion of change that is interested in finding methods calls to define aspect candidates, while our approach is interested in evaluate the current modular structure on the point of view of co-changes.

Adams et al. [2010] proposed a mining technique (COMMIT) to identify concerns

from functions, variables, types, and macros that were changed together. Similarly to HAM, COMMIT is based on the idea that similar calls and references that are added or removed into different parts of the program are candidates to refer to the same concern. This information produces several seed graphs which are concern candidates because nodes in the graph represent program entities to which calls or references have been co-added or co-removed. They enhance the quality of the seed graphs as concern candidates using a filter based on how intentional co-additions and co-removals are. In other words, they quantify how closely related two entities are measuring how often the addition or removal of a dependency on one entity matches the addition or removal of similar dependency on the other. Finally, they rank the seed graphs on the graph dimension and on the scattering. Their technique differs from our technique because they generate independent seed graphs, while our technique is centered on a unique graph.

Mileva and Zeller [2011] mined evolution patterns (changes in the usage of a module) to assess modularity. An evolution pattern is a code change pattern that happened during the software evolution from one version to another. They observed that if a module is present in many evolution patterns, this indicates that its usage had a significant change. Code changes of this module that change its usage are not desired to the module's client. In addition, such changes can lead to potential defects in the module. In contrast, our technique helps to analyze whether software artifacts that usually change together are confined in packages.

## 2.5.5 Hybrid Approaches

Kagdi et al. [2013] combined conceptual and evolutionary couplings for impact analysis in source code, using information retrieval and version history mining techniques. Gethers et al. [2011] proposed an impact analysis that adapts to the specific maintenance scenario using information retrieval, historical data mining, and dynamic analysis techniques. However, they did not use any kind of documentation such as issue reports to discard noisy commits.

Misra et al. [2012] extracted high level component architecture from the underlying source code by combining both syntactic and semantic features. They presented a clustering approach which combines multiple features for determining the strength of relatedness between code elements. In the first instance, they extracted the semantic and syntactic features. Semantic features consist of textual feature, feature based on class names, public method names, and packaging. Syntactic features include feature based on inheritance and dependency. In a further step, a class-to-class similarity es-

timization was computed to estimate a combined similarity score. Next, a clustering process was applied to discover components of the system. The clusters generated and the source code were used to identify public methods of all classes in each cluster that are called by classes in other clusters. These methods represent the interfaces for each cluster. The clusters were automatically labeled through the selection of meaningful words, such as class names and textual features for the classes contained in the cluster. They used the public methods to generate the inter-component interactions. The extracted clusters in the first step were submitted to a clustering hierarchy process. This last clustering process allowed to group the packages instead of classes at the first level. It could be achieved by considering the packages as the clusters of classes and then proceeding further.

Bavota et al. [2014] proposed R3, an automated technique to support re-modularization based on both structural and semantic information extracted from the source code. R3 identifies candidate *move class* refactoring solutions through latent topics in classes and packages and structural dependencies. It uses RTM (Relational Topic Models), a statistic topic modeling technique used to represent and analyzing textual documents and relationship among them [Chang and Blei, 2010]. R3 bases on the RTM similarity matrix to determine the degree of similarity among classes in the system and detect classes similar to a specific class for move class refactoring operation. Their evaluation was based on analysis whether the move class operations suggested by R3 were able to reduce the coupling among packages of nine software systems. They noted that R3 provides a coupling reduction from 10% to 30% among software modules. Their technique differs from our technique because they base on RTM similarity matrix to support re-modularization, while our technique is centered on co-change clusters to assess modularity.

Beck and Diehl [2010] combined evolutionary and syntactic dependencies to retrieve the modular structure of a system. The software clustering approach was used to recover the architecture based on static structural source code dependencies, evolutionary co-change dependencies, and combined structural and evolutionary dependencies. Evolutionary data on software clustering can produce meaningful clustering results, since an extensive historical data is available. However, evolutionary dependencies often cover an insufficient set of classes and it seems to be the major reason structural dependencies overcome the performance of evolutionary data. Finally, their results showed that the integration of both kind of data increases the clustering quality. In contrast, our goal is not recover the architecture but using categorized co-change clusters to support developers for modularity analysis.

Bavota et al. [2013] investigated how different types of coupling aligns with devel-

oper's perceptions. They conducted an empirical study with 76 developers including developers of three open-source projects, students, academic, and industry professionals. They considered coupling measures based on structural, dynamic, semantic, and logical information and evaluate how developers rate the identified coupling links (pairs of classes). Furthermore, they analyzed if the four measures are able to reproduce a modularization similar to the original decomposition. Their results showed that a high number of the coupling relations are captured by semantic and structural measures that seems to complement each other. Moreover, the semantic measure shows to be the most appropriate to reconstruct the original decomposition of a system because it 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, in this thesis we retrieve co-change clusters having at least four classes to evaluate developer's perceptions.

### 2.5.6 Critical Analysis

For years, several attempts have been developed aiming to evaluate software modularity [Beyer and Noack, 2005; Zimmermann et al., 2005; Bavota et al., 2014; Palomba et al., 2013]. Despite modularity being an essential principle in software development, effective approaches to assess modularity still remain an open problem. Table 2.2 summarizes the most closest empirical studies to our work. In this thesis, we propose a modularity assessment technique centered on co-change relations. This technique differs from the presented studies with respect to the follow central aspects:

- Typically, most co-change approaches use association rules; which are extracted from version histories. For instance, to evaluate the feasibility of using association rules to assess modularity, we performed the Apriori<sup>2</sup> algorithm [Agrawal and Srikant, 1994] several times to extract rules for Lucene system.<sup>3</sup> We mined almost one million association rules with minimum support threshold set to four transactions, confidence threshold to 50%, and the maximum size to 10 classes for each rule. Nonetheless, this massive number of rules is a strong limitation to use such technique to assess modularity. Differently, the technique proposed in Chapter 3 relies on sets of co-changes to assess modularity by comparing co-change relations with the system's packages.

---

<sup>2</sup>To execute apriori, we relied on the implementation provided in, <http://www.borgelt.net/apriori.html>.

<sup>3</sup>An information retrieval library, <http://lucene.apache.org>.

**Table 2.2.** Summary of empirical studies on recommendation and modularity analysis.

Study	Goal	Technique	Results	Data Source
[Santos et al., 2014]	Software re-modularization	Semantic clustering	project semantic clusters on distribution maps	Vocabulary
Robillard and Weigand-Warr 2005	Manage concerns and features	Create concern model manually	concern relations Syntactical and structural	Developer
[Beyer and Noack, 2005]	Software visualization	Co-change clustering	clusters of co-change artifacts	Commits
[Zimmermann et al., 2005]	Predict further changes	Association rules	Warning after applying changes	Commits
[Wong et al., 2011]	Detect modularity violations	Association rules and DSM	Identify Discrepancies	Commits and source code
[D’Ambros et al., 2009a]	Visualize co-change information	Association rules	Show logical coupling between modules	Commits
[Kothari et al., 2006]	Characterize the evolution of systems	Change clustering	Classify code change activities	Commits
[Vanya et al., 2008]	Support the partitioning of system	Co-change clustering	Listing of clusters as suggestions	Commits
[Robillard and Dagenais, 2010]	Recommend changes	Change clustering	Selected clusters that matched to a query	Commits
[Palomba et al., 2013]	Detect code smells	Association rules	Detection of five types of code smells	Commits
[Misra et al., 2012]	Extract high level component architecture	Clustering Technique	Clusters labeled automatically	Syntactic and Semantic data
[Bavota et al., 2014]	Support re-modularization	RTM Similarity matrix	Identify candidate <i>move class</i> refactoring	Structural and semantic

- Unlike previous works related to cluster extraction to assess modularity, the technique proposed in this thesis does not extract component architecture or recommend re-modularization. Instead, our technique allows a developer to investigate how often changes in the system are confined in packages and whether changes that crosscut the system’s packages can reveal poor design.

## 2.6 Final Remarks

In this chapter, we presented clustering concepts commonly used in data mining. We emphasized the difference between existent clustering approaches and the contribution of the clustering technique used in this work. After that, we discussed several information retrieval techniques to preprocess textual documents. We focused on the semantic information extracted from issue reports, which are used in Chapter 3 to characterize the semantic similarity of clusters. Next, we provided the concepts of distribution maps and the recurrent patterns which are used in this thesis to compare the actual package modularization with co-change clusters. Finally, we presented the state-of-the-art in version history analysis and approaches based on different software modularity views, such as textual and co-change approaches. Nonetheless, the studies do not investigate whether modules are indeed able to confine changes.





# Chapter 3

## Co-Change Clustering

This chapter is organized as follows. Section 3.1 presents a technique, called Co-Change Clustering to extract co-change graphs and co-change clusters from version control systems. Section 3.2 presents the preliminary results of using co-change clustering, on four systems. Section 3.3 analyzes the modularity of such systems under the light of co-change clusters. Section 3.4 analyzes the semantic similarity within the set of issues related to the extracted clusters. Section 3.5 discusses our results and Section 3.6 presents threats to validity.

### 3.1 Proposed Technique

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

#### 3.1.1 Extracting Co-Change Graphs

In this thesis we rely on the concept of co-change graph proposed by Beyer et al. [Ball et al., 1997; Beyer and Noack, 2005]. Our technique relies on two inputs: issue reports available in issue trackers, e.g., Jira, Bugzilla, and Tigris; and commit transactions retrieved from version control repositories (SVN or GIT). In a further step, several processing tasks are applied and then a co-change graph is build. Finally, sets of classes that frequently change together are retrieved, called co-change clusters.

##### 3.1.1.1 Pre-processing Tasks

When extracting co-change graphs, it is fundamental to preprocess the considered commits to filter out commits that may pollute the graph with noise. We propose the

following preprocessing tasks:

*Removing commits not associated to maintenance issues:* In early implementation stages, commits can denote partial implementations of programming tasks, since the system is under construction [Negara et al., 2012]. When such commits are performed multiple times, they generate noise in the edges’ weights. For this reason, we consider just commits associated to maintenance issues. More specifically, we consider as maintenance issues those that are registered in an issue tracking system. Moreover, we only consider issues labeled as *bug correction*, *new feature*, or *improvement*. We followed the usual procedure to associate commits to maintenance issues: a commit is related to an issue when its textual description includes a substring that represents a valid Issue-ID in the system’s bug tracking system [D’Ambros et al., 2010; Śliwerski et al., 2005; Zimmermann et al., 2007].

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

*Merging commits related to the same maintenance issue:* When there are multiple commits referring to the same Issue-ID, we merge all of them—including the changed classes—in a single commit. Figure 3.1 presents an example for the Geronimo system.<sup>1</sup> The figure shows the short description of four commits related to the issue GERONIMO-3003. In this case, a single change set is generated for the four commits, including 13 classes. In the co-change graph, an edge is created for each pair of classes in this merged change set. In this way, it is possible to have edges connecting classes modified in different commits, but referring to the same maintenance issue.

*Removing commits associated to multiple maintenance issues:* We remove commits that report changes related to more than one maintenance issue, which are usually called tangled code changes [Herzig and Zeller, 2013]. Basically, such commits are discarded because otherwise they would result on edges connecting classes modified to

---

<sup>1</sup>Geronimo is an application server, <http://geronimo.apache.org>.

```

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

```

**Figure 3.1.** Multiple commits for the issue GERONIMO-3003

implement semantically unrelated maintenance tasks (which are included in the same commit just by convenience, for example). Figure 3.2 presents a tangled code change for the Geronimo system.

```

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

```

**Figure 3.2.** Single commit handling multiple issues (3254, 3394 to 3398)

*Removing highly scattered commits:* We remove commits representing highly scattered code changes, i.e., commits that modify a massive number of classes. Typically, such commits are associated to refactorings (like rename method) and other software quality improving tasks (like dead code removal), implementation of new features, or minor syntactical fixes (like changes to comment styles) [Walker et al., 2012]. Figure 3.3 illustrates a highly scattered commit in Lucene. This commit changes 251 classes, located in 80 packages. Basically, in this commit redundant throws clauses are refactored.

Recent research shows that scattering in commits tends to follow heavy-tailed

Revision: 1355069

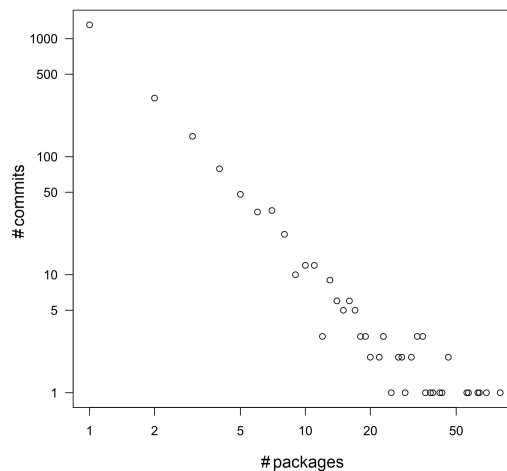
Date: Thu Jun 28 13:39:25 BRT 2012

Short Description: LUCENE-4172: clean up redundant throws clauses

Changed Classes: [251 classes]

**Figure 3.3.** Highly scattered commit (251 changed classes)

distributions [Walker et al., 2012]. Therefore, the existence of massively scattering commits cannot be neglected. Particularly, such commits may have a major impact when considered in co-change graphs, due to the very large deviation between the number of classes changed by them and by the remaining commits in the system. Figure 3.4 illustrates this fact by showing a histogram with the number of packages changed by commits made to the Lucene system.<sup>2</sup> As we can observe, 1,310 commits (62%) change classes in a single package. Despite this fact, the mean value of this distribution is 51.2, due to the existence of commits changing for example, more than 10 packages.



**Figure 3.4.** Packages changed by commits in the Lucene system

Considering that our goal is to model recurrent maintenance tasks and considering that highly scattered commits typically do not present this characteristic, we decided to remove them during the co-change graph creation. For this purpose, we define that a package  $pkg$  is changed by a commit  $cmt$  if at least one of the classes modified by  $cmt$  are located in  $pkg$ . Using this definition, we ignore commits that change more

<sup>2</sup>An information retrieval library, <http://lucene.apache.org>.

than *MAX\_SCATTERING* packages. In Section 3.2, we define and explain the values for thresholds in our method.

### 3.1.1.2 Post-processing Task

In co-change graphs, the edges' weights represent the number of commits changing the connected classes. However, co-change graphs typically have many edges with small weights, i.e., edges representing co-changes that occurred very few times. Such co-changes are not relevant considering that our goal is to model recurrent maintenance tasks. For this reason, there is a post-processing phase after extracting a first co-change graph. In this phase, edges with weights less than a *MIN\_WEIGHT* threshold are removed. In fact, this threshold is analogous to the *support* threshold used by co-change mining approaches based on association rules [Zimmermann et al., 2005].

## 3.1.2 Extracting Co-Change Clusters

After extracting the co-change graph, our goal is to retrieve sets of classes that frequently change together, which we call co-change clusters. We propose to extract co-change clusters automatically, using a graph clustering algorithm designed to handle sparse graphs, as is typically the case of co-change graphs [Beyer and Noack, 2005]. More specifically, we decided to use the Chameleon clustering algorithm, which is an agglomerative and hierarchical clustering algorithm recommended to sparse graphs.

As reported in Section 2.1.2, there are several clustering criterion functions that can be applied in the agglomerative phase available in Cluto package. We conducted pre-experiments with those functions to find which one produces clusters with higher internal similarity, lower external similarity and higher density. The function *i2* (Cohesion based on graphs) is the one that best fitted to our goal. We observed that other functions retrieved several clusters with low density. The *i2* function searches for subclusters to combine, maximizing the similarity by evaluating how close are the objects in a cluster, as follows:

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

### 3.1.2.1 Defining the Number of Clusters

A critical decision when applying Chameleon—and many other clustering algorithms—is to define the number of partitions  $M$  that should be created in the first phase of the

algorithm. To define the “best value” for  $M$  we execute Chameleon multiple times, with different values of  $M$ , starting with a  $M\_INITIAL$  value. Furthermore, in the subsequent executions, the previous tested value is decremented by a  $M\_DECREMENT$  constant.

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

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

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

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

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

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

The proposed  $coefficient(M)$  combines the concepts of cluster cohesion (tight co-change clusters) and cluster separation (highly separated co-change clusters). The  $coefficient$  ranges from  $[-1; 1]$ , where  $-1$  indicates a very poor round and  $1$  an excellent round. The selected  $M$  value is the one with the highest  $coefficient(M)$ . If the highest  $coefficient(M)$  is the same for more than one value of  $M$ , then the highest  $mean(ISim)$  is used as a tiebreaker. Clearly, internal similarity is relevant because it represents how often the classes changed together in a cluster.

## 3.2 Co-Change Clustering Results

In this section, we report the results we achieved after following the methodology described in Section 3.1 to extract co-change clusters for four systems.

**Table 3.1.** Target systems (size metrics)

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

**Table 3.2.** Initial commits sample

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

### 3.2.1 Target Systems and Thresholds Selection

Table 3.1 describes the systems considered in our study, including information on their function, number of lines of code (LOC), number of packages (NOP), and number of classes (NOC). We selected these Java projects because they provide a significant number of commits linked to issue reports, i.e., more evolutionary information to conduct our experiment. Table 3.2 shows the number of commits extracted for each system and the time frame used in this extraction.

In order to run the approach, we define the following thresholds:

- $MAX\_SCATTERING = 10$  packages, i.e., we discard commits changing classes located in more than ten packages. We based on the hypothesis that large transactions typically correspond to noisy data, such as changes in comments formatting and rename method [Zimmermann et al., 2005; Adams et al., 2010]. However, excessive pruning is also undesirable, so we adopted a conservative approach working at package level.
- $MIN\_WEIGHT = 2$  co-changes, i.e., we discard edges connecting classes with less than two co-changes because an unitary weight does not reflect how often two classes usually change together [Beyer and Noack, 2005].
- $M\_INITIAL = NOC_G * 0.20$ , i.e., the first phase of the clustering algorithm creates a number of partitions that is one-fifth of the number of classes in the co-change graph ( $NOC_G$ ). The higher the  $M$ , the higher the final clusters' size because the second phase of the algorithm works by aggregating the partitions. In

this case, the *ISim* tends to be low because subgraphs that are not well connected are grouped in the same cluster. We performed several experiments varying  $M$ 's value, and observed that whenever  $M$  is high, the clustering tends to have clusters of unbalanced size.

- $M\_DECREMENT = 1$  class, i.e., after each clustering execution, we decrement the value of  $M$  by 1.
- $MIN\_CLUSTER\_SZ = 4$  classes, i.e., after each clustering execution, we remove clusters with less than 4 classes.

We defined the thresholds after some preliminary experiments with the target systems. We also based this selection on previous empirical studies reported in the literature. For example, Walker et al. [2012] showed that only 5.93% of the patches in the Mozilla system change more than 11 files. Therefore, we claim that commits changing more than 10 packages are in the last quantiles of the heavy-tailed distributions that normally characterize the degree of scattering in commits. As another example, in the systems included in the Qualitas Corpus—a well-known dataset of Java programs—the packages on average have 12.24 classes [Terra et al., 2013; Tempero et al., 2010]. In our four target systems, the packages have on average 15.87 classes. Therefore, we claim that clusters with less than four classes can be characterized as small clusters.

### 3.2.2 Co-Change Graph Extraction

We start by characterizing the extracted co-change graphs. Table 3.3 shows the percentage of remaining commits in our sample, after applying the preprocessing filters described in Section 3.1.1.1: removal of commits not associated to maintenance issues (Pre #1), removal of commits not changing classes and also referring to testing classes (Pre #2), merging commits associated to the same maintenance issue (Pre #3), removal of commits denoting tangled code changes (Pre #4), and removal of highly scattering commits (Pre #5).

As can be observed in Table 3.3, our initial sample for the Geronimo, Lucene, JDT Core, and Camel systems was reduced to 14.3%, 22.4%, 20.1% and 21.3% of its original size, respectively. The most significant reduction was due to the first preprocessing task. Basically, only 32.6%, 39.2%, 38.4%, and 45.0% of the commits in Geronimo, Lucene, JDT Core, and Camel are associated to maintenance issues (as stored in the systems issue tracking platforms). Moreover, we analyzed the commits discarded in first preprocessing task. We observed a substantial number of commits changing a



**Table 3.3.** Percentage of unitary commits (i.e., changing a single class) discarded in the first phase and commits discarded after each preprocessing filters

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

single class, 39.6% of Geronimo’s, 35.3% of Lucene’s, 58.1% of JDT’s, and 44.5% of Camel’s commits. These unitary commits may contain configuration or/and script files, for instance. In addition, as some of these commits are not linked to issue reports, we cannot assume they represent a complete maintenance task. We also had not analyzed if these unitary commits are partial implementations of maintenance tasks. This could be done by inspecting their time frame, for instance. However, these unitary commits are not useful anyway to evaluate a system in terms of co-changes. There are also significant reductions after filtering out commits that do not change classes or that only change testing classes (preprocessing task #2) and after merging commits related to the same maintenance issue (preprocessing task #3). Finally, a reduction affecting 3% of the Geronimo’s commits, 4% of the Camel’s commits, and nearly 1% of the commits of the other systems is observed after the last two preprocessing tasks.

After applying the preprocessing filters, we extracted a first co-change graph for each system. We then applied the post-processing filter defined in Section 3.1.1.2, to remove edges with unitary weights. Table 3.4 shows the number of vertices ( $|V|$ ) and the number of edges ( $|E|$ ) in the co-change graphs, before and after this post-processing task. The table also presents the graph’s density (column D).

**Table 3.4.** Number of vertices ( $|V|$ ), edges ( $|E|$ ) and co-change graphs’ density (D) before and after the post-processing filter

System	<b>Post-Processing</b>					
	<b>Before</b>			<b>After</b>		
	$ V $	$ E $	D	$ V $	$ E $	D
Geronimo	2,099	24,815	0.01	695	4,608	0.02
Lucene	2,679	63,075	0.02	1,353	18,784	0.02
JDT Core	1,201	75,006	0.01	823	25,144	0.04
Camel	3,033	42,336	0.01	1,498	15,404	0.01

By observing the results in Table 3.4, two conclusions can be drawn. First, co-change graphs are clearly sparse graphs, having density close to zero in the evaluated

systems. This fact reinforces our choice to use Chameleon as the clustering algorithm, since this algorithm is particularly well-suited to handle sparse graphs Karypis et al. [1999]. Second, most edges in the initial co-change graphs have weight equal to one (more precisely, around 81%, 70%, 66%, and 64% of the edges for Geronimo, Lucene, JTD Core, and Camel graphs, respectively). Therefore, these edges connect classes that changed together in just one commit and for this reason they were removed after the post-processing task. As result, the number of vertices after post-processing is reduced to 33.1% (Geronimo), 50.5% (Lucene), 68.5% (JTD Core), and 49.4% (Camel) of their initial value.

### 3.2.3 Co-Change Clustering

We executed the Chameleon algorithm having as input the co-change graphs created for each system (after applying the pre-processing and post-processing filters).<sup>3</sup> Table 3.5 shows the value of  $M$  that generated the best clusters, according to the clustering selection criteria defined in Section 3.1.2.1. The table also reports the initial number of co-change clusters generated by Chameleon and the number of clusters after eliminating the small clusters, i.e., clusters with less than four classes, as defined by the *MIN\_CLUSTER\_SZ* threshold. Finally, the table shows the ratio between the final number of clusters and the number of packages in each system (column %NOP).

**Table 3.5.** Number of co-change clusters

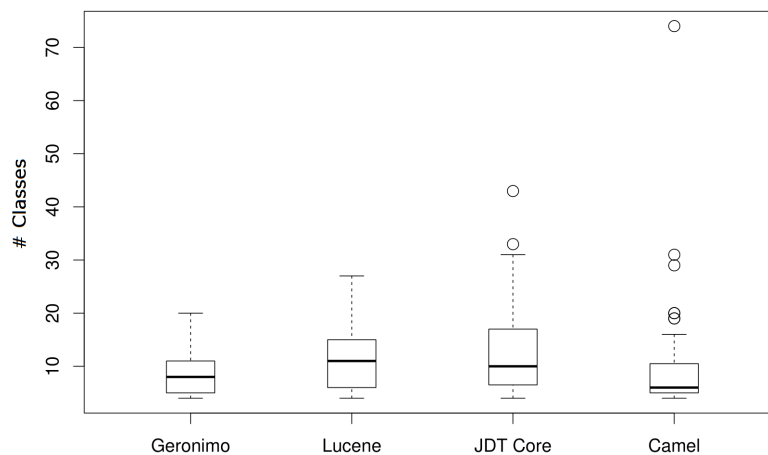
System	M	# clusters		%NOP
		All	$ V  \geq 4$	
Geronimo	108	46	21	0.05
Lucene	68	98	49	0.19
JTD Core	100	35	24	0.32
Camel	251	130	47	0.06

For example, for Geronimo, we achieved the “best clusters” for  $M = 108$ , i.e., the co-change graph was initially partitioned into 108 clusters, in the first phase of the algorithm. In the second phase (agglomerative clustering), the initial clusters were successively merged, stopping with a configuration of 46 clusters. However, only 21 clusters have four or more classes ( $|V| \geq 4$ ) and the others were discarded, since they represent “small modules”, as defined in Section 3.2.1. We can also observe that the number of clusters is considerably smaller than the number of packages. Basically, this

<sup>3</sup>To execute Chameleon, we relied on the CLUTO clustering package, <http://glaros.dtc.umn.edu/gkhome/cluto/cluto/overview>.

fact is an indication that the maintenance activity in the systems is concentrated in few classes.

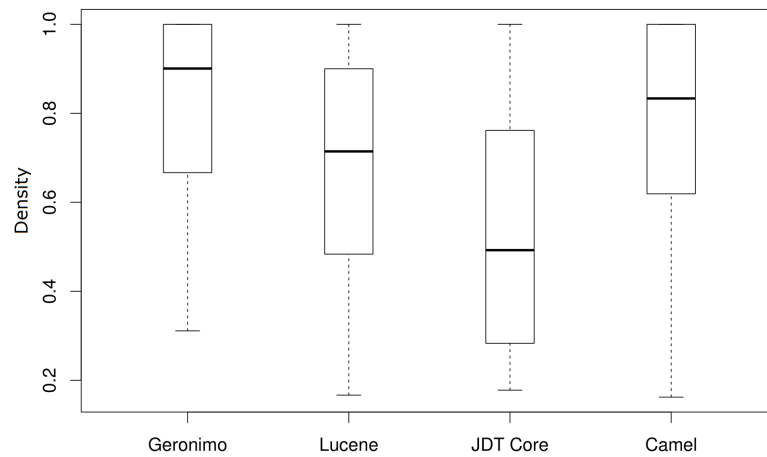
Figure 3.5 shows the distribution regarding the size of the extracted co-change clusters for each system, in terms of number of classes. The extracted clusters have  $8.8 \pm 4.7$  classes,  $11.7 \pm 7.0$  classes,  $14 \pm 10.4$  classes, and  $10.2 \pm 11.48$  (average  $\pm$  standard deviation) in the Geronimo, Lucene, JDT Core, and Camel systems, respectively. Moreover, the median size is 8 (Geronimo), 11 (Lucene), 10 (JDT Core), and 6 (Camel) and the biggest cluster has a considerable number of classes: 20 classes (Geronimo), 27 classes (Lucene), 43 classes (JDT Core), and 74 classes (Camel).



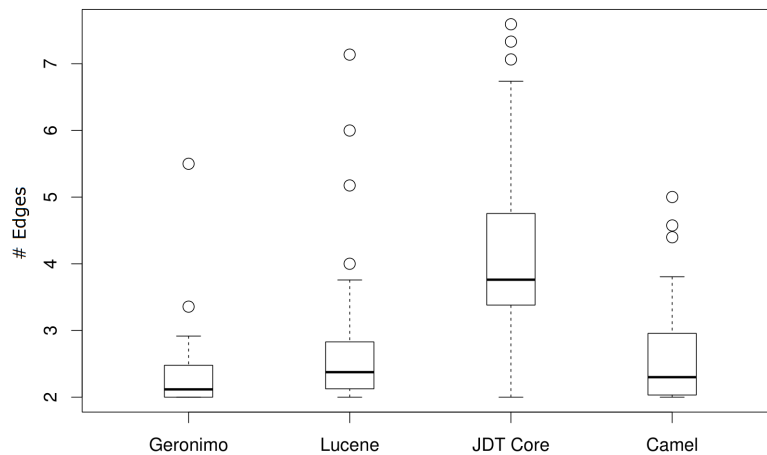
**Figure 3.5.** Co-change clusters size (in number of classes)

Figure 3.6 presents the distribution of the densities of the co-change clusters extracted for each system.. The clusters have density of  $0.80 \pm 0.24$  (Geronimo),  $0.68 \pm 0.25$  (Lucene),  $0.54 \pm 0.29$  (JDT Core), and  $0.77 \pm 0.25$  (Camel). The median density is 0.90 (Geronimo), 0.71 (Lucene), 0.49 (JDT Core), and 0.83 (Camel). Therefore, although co-change graphs are sparse graphs, the results in Figure 3.6 show they have dense subgraphs with a considerable size (at least four classes). Density is a central property in co-change clusters, because it assures that there is a high probability of co-changes between each pair of classes in the cluster.

Figure 3.7 presents the distribution regarding the average weight of the edges in the extracted co-change clusters for each system. For a given co-change cluster, we define this average as the sum of the weights of all edges divided by the number of edges in the cluster. We can observe that the median edges' weight is not high, being slightly greater than two in Geronimo, Lucene, and Camel. Whereas, in the JDT Core it is about four.



**Figure 3.6.** Co-change clusters density



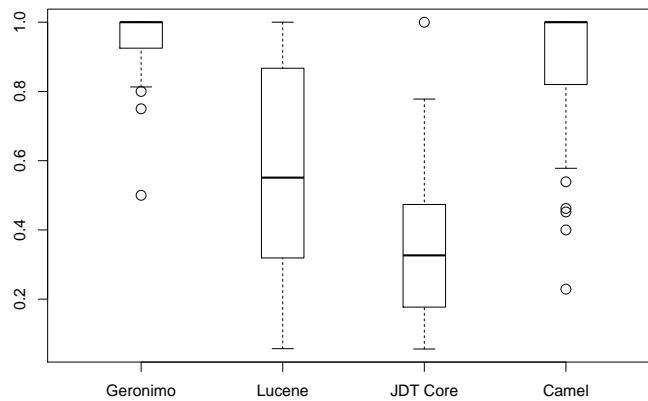
**Figure 3.7.** Cluster average edges' weight

### 3.3 Modularity Analysis

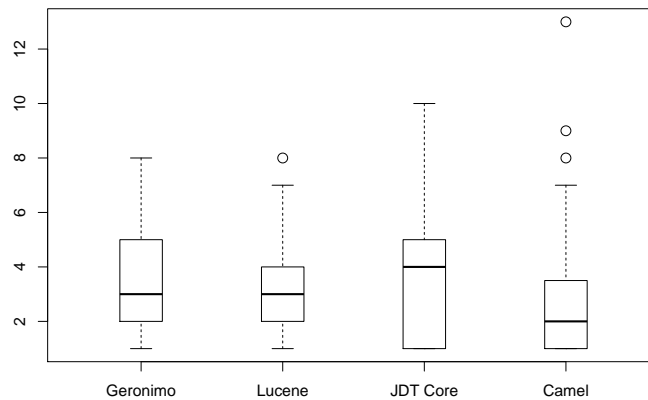
In this section, we investigate the application of co-change clusters to assess the quality of a system's package decomposition. Particularly, we investigate the distribution of the co-change clusters over the package structure. For this purpose, we rely on distribution maps [Ducasse et al., 2006], which are a well-known visualization and comprehension technique.

Figures 3.8 and 3.9 show the distribution regarding focus and spread of the co-change clusters for each system. We can observe that the co-change clusters in Geronimo and Camel have a higher focus than in Lucene and JDT Core. For example, the median focus in Geronimo and Camel is 1.00, against 0.55 and 0.30 in Lucene and

JDT Core, respectively. Regarding spread, Camel has a lower value than the others, on average the spread is 2.96 against 3.50 (Geronimo), 3.35 (Lucene), and 3.83 (JDT Core). Figure 3.10 shows a scatterplot with the values of focus (horizontal axis) and spread (vertical axis) for each co-change cluster. In Geronimo and Camel, we can see that there is a concentration of clusters with high focus. On the other hand, for Lucene, the clusters are much more dispersed along the two axis. Eclipse JDT tends to have lower focus, but also lower spread.



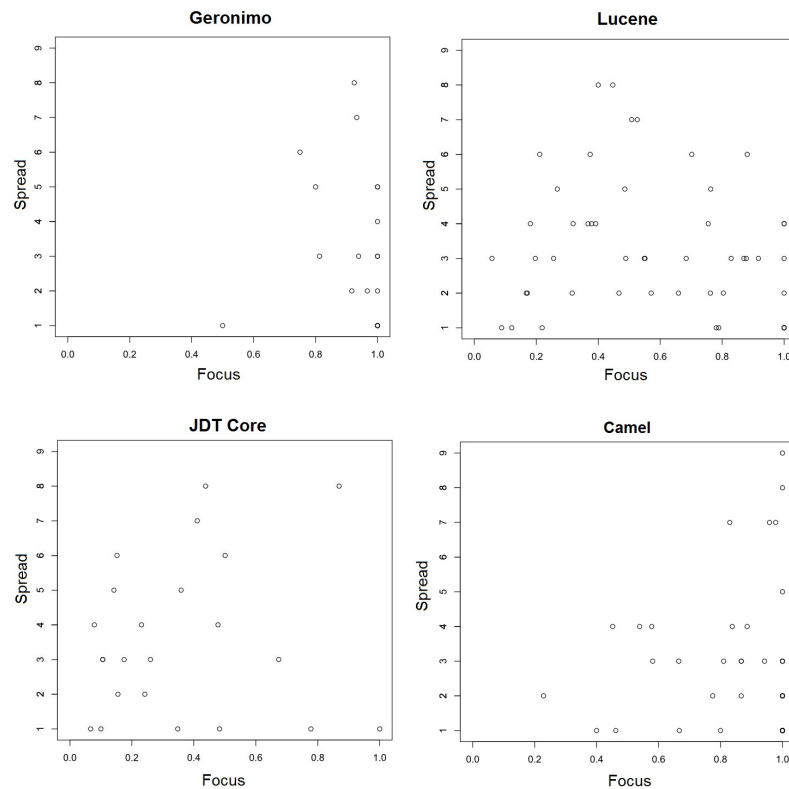
**Figure 3.8.** Focus



**Figure 3.9.** Spread

In the following sections, we analyze examples of clusters which dominate the packages they touch and clusters scattered over packages, using distribution maps.<sup>4</sup> Section 3.3.1 emphasizes clusters with high focus (focus  $\approx$  or  $=$  1.0), since they are

<sup>4</sup>To extract and visualize distribution maps, we used the Topic Viewer tool [Santos et al., 2014], available at <https://code.google.com/p/topic-viewer>.



**Figure 3.10.** Focus versus Spread

common in Geronimo. On the other hand, Section 3.3.2 emphasizes scattered concerns, which are most common in Lucene. Section 3.3.3 reports on both types of clusters in Eclipse JDT. Finally, analogous to Geronimo, Section 3.3.4 emphasizes clusters with high focus, since they are common in Camel.

### 3.3.1 Distribution Map for Geronimo

Figure 3.11 shows the distribution map for Geronimo. To improve the visualization, besides background colors, we use a number in each class (small squares) to indicate their respective clusters. The large boxes are the packages and the text below is the package name.

Considering the clusters with high focus in Geronimo, we found three package distribution cases:

- *Clusters that touch a single package (spread = 1) and dominate it (focus = 1.0).* Four clusters have this behavior. As an example, we have Cluster 2, which dominates the co-change classes in the package `main.webapp.WEBINF.view.-realmwizard` (line 2 in the map, column 3). This package implements a wizard

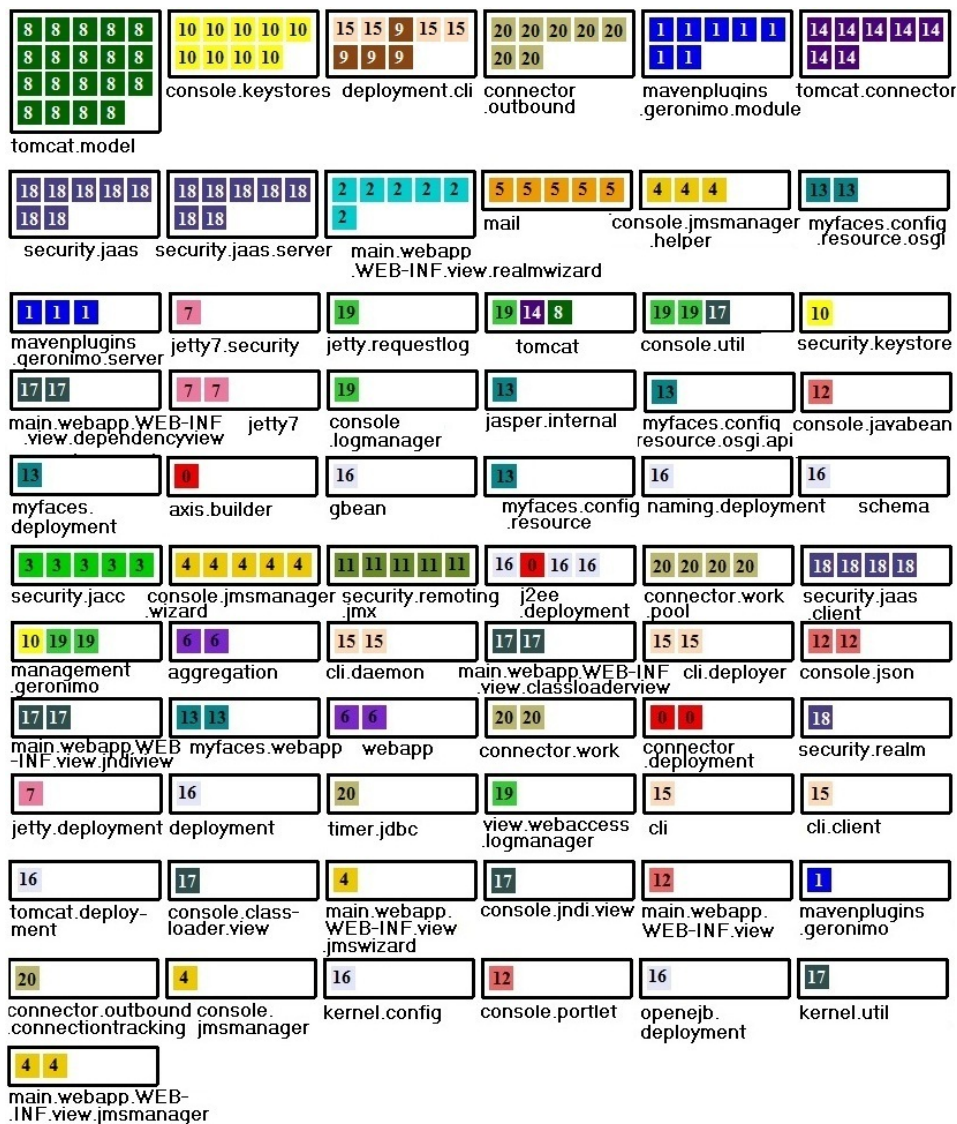


Figure 3.11. Distribution map for Geronimo

to configure or create security domains. Therefore, since it implements a specific functional concern, maintenance is confined in the package. As another example, we have Cluster 5 (package `mail`, line 2 in the map, column 4) and Cluster 11 (package `security.remoting.jmx`, line 6, column 3).

- Clusters that dominate the packages ( $focus = 1.0$ ) they touch ( $spread > 1$ ). We counted eight clusters with this behavior. As an example, we have Cluster 18 ( $spread = 4$ ), which touches all co-change classes in the following packages: `security.jaas.server`, `security.jaas.client`, `security.jaas`, and `security.realm` (displayed respectively in line 2, columns 1 and 2; line 6, col-

umn 6; and line 8, column 6). As suggested by their names, these packages are related to security concerns, implemented using the Java Authentication and Authorization Service (JAAS) framework. Therefore, the packages are conceptually related and their spread should not be regarded as a design problem. In fact, the spread in this case is probably due to a decision to organize the source code in sub-packages. As another example, we have Cluster 20 ( $spread = 5$ ), which touches all classes in `connector.outbound`, `connector.work.pool`, `connector.work`, `connector.outbound.connectiontracking`, and `timer.jdbc` (displayed respectively in line 1, column 4; line 6, column 5; line 8, column 4; line 9, column 3; line 11 and column 1). These packages implement EJB connectors for message exchange.

- *Clusters that dominate a package partially ( $focus \approx 1.0$ ) and touching some classes in other packages ( $spread > 1$ ).*<sup>5</sup> As an example, we have Cluster 8 ( $focus = 0.97$ ,  $spread = 2$ ), which dominates the co-change classes in the package `tomcat.model` (line 1 and column 1 in the map), but also touches the class `TomcatServerGBean` from package `tomcat` (line 3, column 4). This class is responsible for configuring the web server used by Geronimo (Tomcat). Therefore, this particular co-change instance suggests an instability in the interface provided by the web server. In theory, Geronimo should only call this interface to configure the web server, but the co-change cluster shows that maintenance in the `model` package sometimes has a ripple effect on this class, or vice-versa. As another example, we have Cluster 14 ( $focus = 0.92$  and  $spread = 2$ ), which dominates the package `tomcat.connector` (line 1 and column 6 in the map) but also touches the class `TomcatServerConfigManager` from package `tomcat` (line 3, column 4). This “tentacle” in a single class from another package suggests again an instability in the configuration interface provided by the underlying web server.

### 3.3.2 Distribution Map for Lucene

We selected for analysis clusters that are *scattered* ( $focus \approx 0.0$ ), since they are much more common in Lucene. More specifically, we selected the three clusters in Lucene with the lowest focus and a spread greater than two. Figure 3.12 shows a fragment of the distribution map for Lucene, containing the following clusters:

---

<sup>5</sup>These clusters are called octopus, because they have a body centered on a single package and tentacles in other packages [Ducasse et al., 2006].



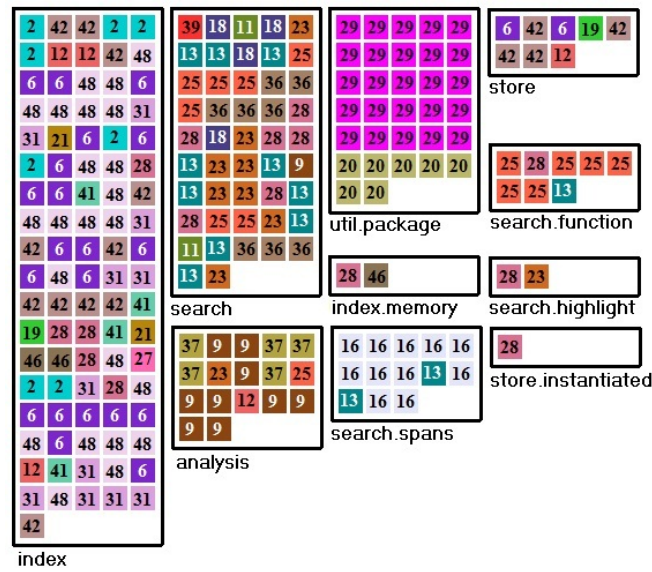


Figure 3.12. Part of the Distribution map for Lucene

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

We also analyzed the maintenance issues associated to the commits responsible for the co-changes in Cluster 28. Particularly, we retrieved 37 maintenance issues related to

this cluster. We then manually read and analyzed the short description of each issue, and classified them in three groups: (a) maintenance related to functional concerns in Lucene’s domain (like searching, indexing, etc); (b) maintenance related to non-functional concerns (like logging, persistence, exception handling, etc); (c) maintenance related to refactorings. Table 3.6 shows the number of issues in each category. As can be observed, the crosscutting behavior of Cluster 28 is more due to issues related to functional concerns (59.5%) than to non-functional concerns (8%). Moreover, changes motivated by refactorings (32.5%) are more common than changes in non-functional concerns.

**Table 3.6.** Maintenance issues in Cluster 28

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

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

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

### 3.3.3 Distribution Map for JDT Core

Figure 3.13 shows the distribution map for JDT Core. We selected three distinct types of clusters for analysis: a *scattered* cluster (*focus*  $\approx 0.0$  and *spread*  $\geq 3$ ), clusters confined in a single package, and a cluster with high spread.

- *Clusters with crosscutting behavior.* We have Cluster 4 (*focus* = 0.08 and *spread* = 4) with co-change classes in the following packages: `internal.compiler.lookup`, `internal.core`, `core.dom`, and `internal.core.util`. The `core.util` package provides a set of tools and utilities for manipulating `.class` files and Java model elements. Since the cluster crosscuts packages providing different services

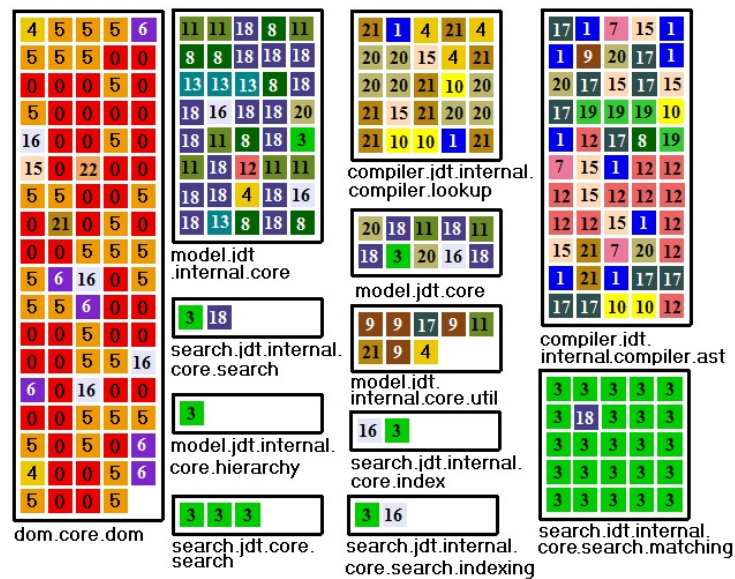


Figure 3.13. Part of the Distribution map for JDT Core

(document structure, files and elements manipulation, population of the model, compiler infrastructure), we claim that it reveals a modularization flaw in the system.

- *Clusters confined in a single package (spread = 1)*. We have Cluster 0 ( $focus = 0.48$ ), Cluster 5 ( $focus = 0.35$ ), and Cluster 6 ( $focus = 0.07$ ) in the `core.dom` package (line 1, column 1).
- *Clusters that dominate a package partially (focus  $\approx 1.0$ ) and touching some classes in other packages (spread > 1)*. We have Cluster 3 ( $focus = 0.87$  and  $spread = 8$ ), which dominates the co-change classes in the packages `search.jdt.internal.core.search.matching` and `search.jdt.core.search`. These packages provide support for searching the workspace for Java elements matching a particular description. Therefore, a maintenance in this of the cluster usually has a ripple effect in classes like that.

### 3.3.4 Distribution Map for Camel

Figure 3.14 shows the distribution map for Camel. We selected two types of cluster for analysis: a cluster that dominates the packages it touches ( $focus = 1.0$ ) and a partially dominant cluster ( $focus \approx 1.0$ ).

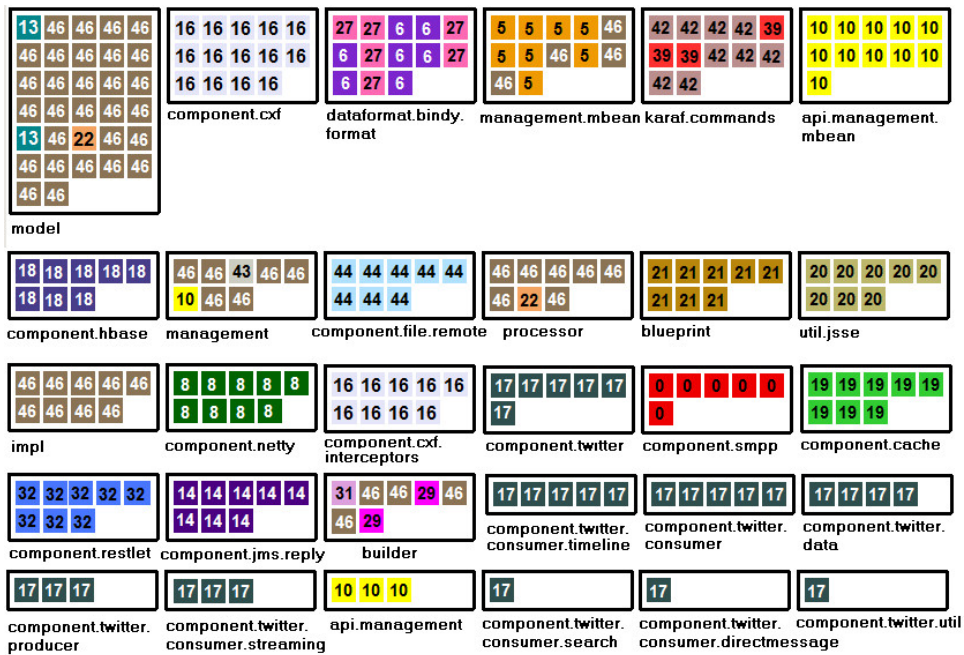


Figure 3.14. Distribution map for Camel

- *Clusters that touch a single package (spread = 1) and dominate it (focus = 1.0)*. Twelve clusters have this behavior. As an example, we have Cluster 0, which dominates the co-change classes in the package `component.smpp` (line 3 in the map, column 5). This package provides access to a short message service. Therefore, since it implements a particular functional concern, the maintenance is confined in the package.
- *Clusters that dominate the packages (focus = 1.0) they touch (spread > 1)*. We counted 13 clusters with this behavior. As an example, we have Cluster 17 (*spread* = 9), which touches all co-change classes in the following packages: `component.twitter`, `component.twitter.data`, `component.twitter.producer`, `component.twitter.consumer.streaming`, `component.twitter.consumer.directmessage`, `component.twitter.consumer.search`, `component.twitter.util`, `component.twitter.consumer`, and `component.twitter.consumer.timeline`. As revealed by their names, these packages are related to the Twitter API. Therefore, these packages are conceptually related and their spread should not be regarded as a design problem. In fact, the spread in this case is probably due to a decision to organize the code in sub-packages.

- *Clusters that dominate a package partially ( $focus \approx 1.0$ ) and touching some classes in other packages ( $spread > 1$ ).* We counted 11 clusters with this behavior. As an example, we have Cluster 10 ( $focus = 0.94$ ,  $spread = 3$ ), which dominates the co-change classes in the packages `api.management` and `api.management.mbean` (line 5, column 3; line 1, column 6 in the map), but also touches the class `MBeanInfoAssembler` from package `management` (line 2, columns 2). This class is responsible for reading details from different annotations, such as `ManagedResource` and `ManagedAttribute`. This co-change cluster shows that maintenance in `api.management` and `api.management.mbean` packages sometimes have a ripple effect on this class, or vice-versa.

## 3.4 Semantic Similarity Analysis

The previous section showed that the package structure of Geronimo and Camel has more adherence to co-change clusters than Lucene's and JDT Core's. We also observed that patterns followed by the relation clusters vs. packages can help to assess the modularity of systems. This section aims at evaluating the semantic similarity of the issues associated to a specific cluster in order to improve our understanding of the clusters' meaning. We hypothesize that if the issues related to a cluster have high semantic similarity, then the classes within that cluster are also semantically related and the cluster is semantically cohesive. We assume that an issue is related to a cluster if the change set of the issue contains at least a pair of classes from that cluster, not necessarily linked with an edge. In our strategy to evaluate the similarity of the issues related to a cluster, we consider each short description of an issue as a document and the collection of documents is obtained from the collection of issues related to a cluster. We use Latent Semantic Analysis - LSA [Deerwester et al., 1990]—presented in Section 2.2—to evaluate the similarity among the collection of documents related to a cluster because it is a well-known method used in other studies concerning similarity among issues and other software artifacts [Poshyvanyk and Marcus, 2008], [Poshyvanyk and Marcus, 2007].

### 3.4.1 Pre-processing Issue Description

When analyzing text documents with Information Retrieval techniques, an adequate pre-processing of the text is important to achieve good results. We define a domain vocabulary of terms based on words found in commits of the target system. The first step is stemming the terms. Next, the stop-words are removed. The final step

produces a term-document matrix, where the cells have value 1 if the term occurs in the document and 0 otherwise. This decision was taken after some qualitative experimentation, in which we observed that different weighting mechanisms based on the frequency of terms, such as td-idf [Manning et al., 2008], did not improve the quality of the similarity matrix.

### 3.4.2 Latent Semantic Analysis

The LSA algorithm is applied to the binary term-document matrix and produces another similarity matrix among the documents (issues) with values ranging from -1 (no similarity) to 1 (maximum similarity). The LSA matrix should have high values to denote a collection of issues that are all related among them. However, not all pairs of issues have the same similarity level, so it is necessary to analyze the degree of similarity between the issues to evaluate the overall similarity within a cluster. We used heat maps to visualize the similarity between issues related to a cluster. Figure 3.15 shows examples of similarity within specific clusters. We show for each system the two best clusters in terms of similarity in the left, and the two clusters with several pairs of issues with low similarity in the right. The white cells represent issues that do not have any word in common, blue cells represent very low similarity, and yellow cells denote the maximum similarity between the issues. We can observe that even for the cluster with more blue cells, there is still a dominance of higher similarity cells. The white cells in JDT's clusters suggest that there are issues with no similarity between the others in their respective cluster.

### 3.4.3 Scoring clusters

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

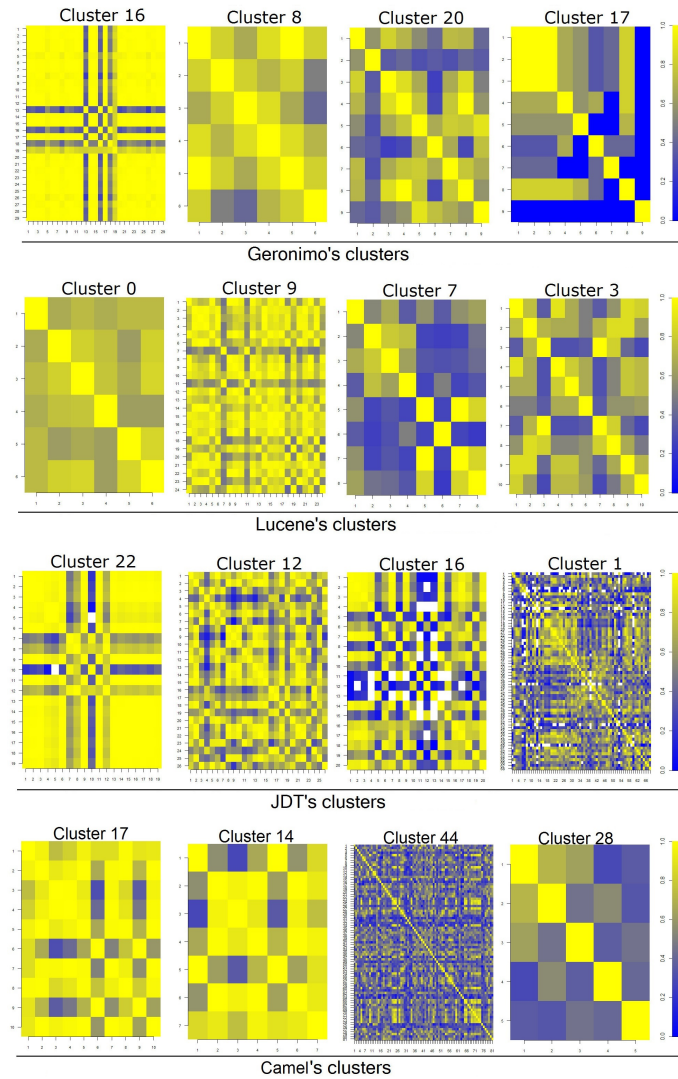
$$similarity\ score(c) = \frac{\sum_{\substack{0 < i, j < n-1 \\ j < i}} similar(i, j)}{\left(\frac{n^2}{2} - n\right)}$$

where

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

$n$  = number of issues related to cluster  $c$

$SIM\_THRS = 0.4$



**Figure 3.15.** Examples of heat maps for similarity of issues

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

In this work, we defined a threshold to evaluate if two issues are similar or not. We consider the semantic similarity between two issue reports,  $i$  and  $j$ , as the cosine between the vectors corresponding to  $i$  and  $j$  in the semantic space created by LSA. After experimental testing, we observed that pairs of issues  $(i, j)$  that had  $LSA\_Cosine(i, j) \geq 0.4$  had a meaningful degree of similarity. Nonetheless, we agree that this fixed threshold is not free of imprecision. Similar to our study, Poshyvanyk and Marcus [2008] used LSA to analyze the coherence of the user comments in bug re-

ports. The system’s developers classified as high/very high similar the comments with average similarity greater than 0.33. For this reason, our more conservative approach seems to be adequate. Moreover, because our goal is to provide an overall evaluation of the whole collection of co-change clusters, some imprecision in the characterization of similarity between two issues would not affect significantly our analysis.

Figure 3.16 shows the distribution of score values for Geronimo’s, Lucene’s, JDT’s, and Camel’s clusters. We can observe that the systems’ clusters follow a similar pattern of scoring, with 100% (for Lucene, JDT, and Camel) and more than 90% (for Geronimo) of clusters having more than half pairs of issues similar to each other. Only two Camel’s clusters have score less than 50% of similarity. Interestingly, one of these two clusters have 226 issue reports and their similarity is very low.

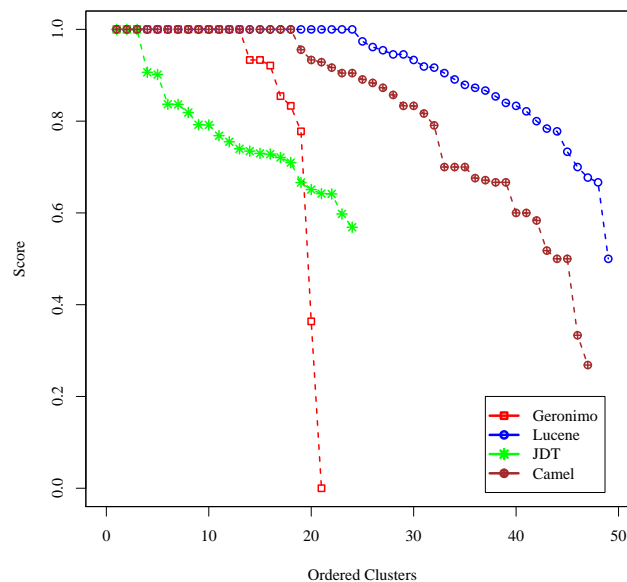


Figure 3.16. Distribution of the clusters’ score

### 3.4.4 Correlating Similarity, Focus, and Spread

Another analysis that we carried out with clusters’ scores was to evaluate the degree of correlation between the score, focus and spread. Table 3.7 shows the results obtained by applying the Spearman correlation test. For Geronimo, we observed a strong negative correlation between spread and score. In other words, the higher is the number of similar issues in a cluster, the higher is the capacity of the cluster to encompass a whole package in Geronimo. Interestingly, Lucene does not present the same behavior



as Geronimo. We observe a weak correlation between focus and score, but we encounter no significant correlation between spread and score. In the case of Lucene, the higher is the number of similar issues in a cluster, the lower is the number of packages touched by the cluster. In the case of Eclipse JDT Core, there is no significant correlation between focus and score. Although, there is a moderate negative correlation between spread and score, it is only significant at p-value 0.074. For Camel, we observed a moderate negative correlation between spread and score. Similar to Geronimo, the higher is the number of similar issues in a cluster, the higher is the capacity of the cluster to enfold a whole package in Camel. Considering that the clusters of the analyzed systems follow a similar pattern of similarity, this result suggests that the similarity between co-changes induces different properties in the clusters, either in spread or in focus.

**Table 3.7.** Correlation between score, focus and spread of clusters for Geronimo, Lucene, JDT Core, and Camel

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

## 3.5 Discussion

### 3.5.1 Practical Implications

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

- When the package structure is adherent to the cluster structure, as in Geronimo’s and Camel’s clusters, localized co-changes are likely to occur.
- When there is not a clear adherence between co-change clusters and packages, a restructuring of the package decomposition may improve modularity. Particularly, there are two patterns of clusters that may suggest modularity flaws. The first pattern denotes clusters with crosscutting behavior (focus  $\approx 0$  and high spread). For example, in Lucene and JDT Core, we detected 12 and 10 clusters related to this pattern, respectively. The second pattern is the octopus cluster

that suggest a possible ripple effect during maintenance tasks. In Geronimo, Lucene, and Camel, we detected four, five, and eleven clusters related to this pattern, respectively.

Nonetheless, we have no evidence that the proposed co-change clusters can fully replace traditional modular decompositions. Indeed, a first obstacle to this proposal is the fact that co-change clusters do not cover the whole population of classes in a system. However, we believe that they can be used as an alternative modular view during program comprehension tasks. For example, they may provide a better context during maintenance tasks (similar for example to the task context automatically inferred by tools like Mylyn [Kersten and Murphy, 2006]).

### 3.5.2 Clustering vs Association Rules Mining

Our technique is centered on the Chameleon hierarchical clustering algorithm, which is an algorithm designed to handle sparse graphs [Karypis et al., 1999]. In our case studies, for example, the co-change graphs have densities ranging from 1% (Camel) to 4% (Eclipse JDT Core).

Particularly, in traditional clustering algorithms, like K-Means [MacQueen, 1967], the mapping of data items to clusters is a total function, i.e., each data item is allocated to a specific cluster. Likewise K-Means, Chameleon tries to cluster all data items. However, it is possible that some vertices are not allocated to any cluster. This may happen when some vertices do not share any edge with the rest of the vertices or when a vertice share edges to other vertices that belong to different clusters with no significant discrepancy among weights.

We also performed an algorithm to detect communities (clusters) [Blondel et al., 2008], provided by the Gephi Tool<sup>6</sup>, to compare with our co-change clusters. Similar to K-Means, this algorithm also allocates each vertice to a particular cluster. Thus, vertices with few or even one edge are assigned to a cluster leading these clusters to have lower density than Chameleon's. Nonetheless, we could define a pos-processing task to prune such vertices from clusters detected by the community algorithm to increase their densities. In spite of clusters with lower density, the result suggested the same pattern we presented in this section, e.g., for Geronimo the package structure is adherent to the cluster structure and for Lucene, there is not a clear adherence between co-change clusters and packages.

---

<sup>6</sup><http://gephi.github.io/>.

An alternative to retrieve co-change relations is to rely on association rules mining [Agrawal and Srikant, 1994]. In the context of evolutionary coupling, an association rule  $C_{ant} \Rightarrow C_{cons}$  expresses that commit transactions changing the classes  $C_{ant}$  (antecedent term) also change  $C_{cons}$  classes (consequent term), with a given probability.

However, hundreds of thousands of association rules can be easily retrieved from version histories. For example, we executed the Apriori<sup>7</sup> algorithm [Agrawal and Srikant, 1994] to retrieve association rules on Lucene’s pre-processed dataset. By defining a minimum support threshold of four transactions, a minimum confidence of 50%, and limiting the size of the rules to 10 classes, we mined 976,572 association rules, with an average size of 8.14 classes. We repeated this experiment with the confidence threshold of 90%. In this case, we mined 831,795 association rules, with an average size of 8.23 classes. This explosion in the number of rules is an important limitation for using association rules to assess modularity, which ultimately is a task that requires careful judgment and analysis by software developers and maintainers. Another attempt to reduce the number of rules is to select the more interesting ones. There are several alternative measures available to complement the support and confidence measures [Piatetsky-Shapiro, 1991], [Omiecinski, 2003]. One of the most well-known is the lift [Brin et al., 1997]. However, if the rules present high values of lift, it is very hard to make a precise selection. Another way to reduce the number of rules is to combine association rules and clustering [Lent et al., 1997].

## 3.6 Threats to Validity

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

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

*Threats to Construct Validity:* A possible design threat to construct validity is that developers might not adequately link commit with issues, as pointed out by Herzig and

---

<sup>7</sup>To execute apriori, we relied on the implementation provided in, <http://www.borgelt.net/apriori.html>.

Zeller [2013]. Moreover, we also found a high number of commits not associated to maintenance issues. Thus, our results are subjected to missing and to incorrect links between commits and issues. However, we claim that we followed the approach commonly used in other studies that map issues to commits [D’Ambros et al., 2010],[Zimmermann et al., 2007], [Couto et al., 2012], [Couto et al., 2014]. We also filtered out situations like commits associated to multiple maintenance issues and highly scattered commits. Another possible construction threat concerns the time frame used to collect the issues. We considered maintenance activity during a period of approximately ten years, which is certainly a large time frame. However, we did not evaluate how the co-change clusters evolved during this time frame or whether the systems’ architecture has changed.

Finally, our technique only handles co-changes related to source code artifacts (.java files). However, the systems we evaluated have other types of artifacts, like XML configuration files. Geronimo for example has 177 Javascript files, 1,004 XML configuration files, 19 configuration files, and 105 image files. Therefore, it is possible that we missed some co-change relations among non-Java based artifacts or between non-Java and Java-based artifacts. However, considering only source code artifacts makes possible the projection of co-change clusters to distribution maps, using the package structure as the main partition in the maps.

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

## 3.7 Final Remarks

In this chapter, we presented technique to extract an alternative view to the package decomposition based on co-change clusters. We applied our technique to four real software systems (Geronimo, Lucene, JDT Core, and Camel). Our results show

that meaningful co-change clusters can be extracted using the information available in version control systems. Although co-change graphs extracted from repositories are sparse, the co-change clusters are dense and have high internal similarity concerning co-changes and semantic similarity concerning their originating issues. We showed that co-change clusters and their associated metrics are useful to assess the modular decomposition of the evaluated systems.



# Chapter 4

## Co-Change Patterns

In this chapter, we start first by characterizing co-change patterns commonly detected in co-change clusters (Section 4.1). In Section 4.2, we present `ModularityCheck`, a tool for assessing modularity. We also present an example of usage, where `ModularityCheck` computes co-change clusters, present metrics values, and categorize recurrent patterns in Geronimo system.

### 4.1 Proposed Co-Change Patterns

In this section, we describe in detail six co-change patterns—five patterns were borrowed from distribution map technique, see Section 2.3—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 contain classes as small squares in large rectangles, which represent packages. The color of the classes represent a property; in our specific case, the co-change cluster.

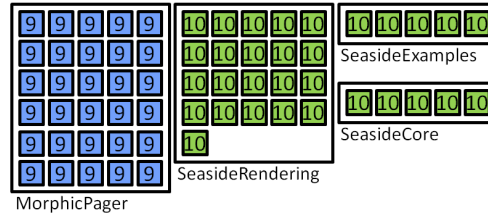
Co-change patterns are defined using two metrics originally proposed for distribution maps: *focus* and *spread*. First, *spread* measures how many packages are touched by a cluster  $q$ . Second, *focus* measures the degree the classes in a co-change cluster dominate their packages. For example, if a cluster touches all classes of a package, its focus is one. The formal terms are presented in Section 2.3.

Using focus and spread, we describe six patterns of co-change clusters, as follows:

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

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

Figure 4.1 shows two examples of Encapsulated Clusters.<sup>1</sup> 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.

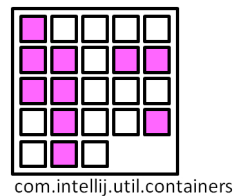


**Figure 4.1.** Encapsulated clusters (Glamour)

*Well-Confined:* Conceptually, the cluster  $q$  touches a single package and does not dominate it. In other words, a co-change cluster  $q$  is categorized as Well-Confined Cluster if:

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

Figure 4.2 shows a Well-Confined Cluster, this cluster touches a single package and shares the package it touches with other clusters (its focus is 0.43).



**Figure 4.2.** Well-Confined cluster (Intellij-Community)

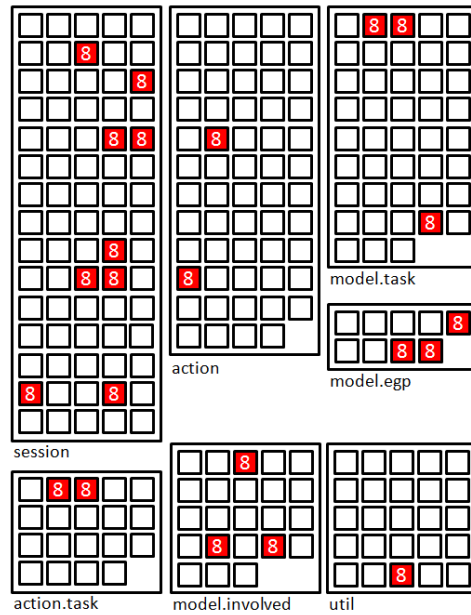
*Crosscutting:* Conceptually, a Crosscutting Cluster is spread over several packages but touches few classes in each one. In practical terms, we propose the following thresholds to represent a Crosscutting cluster  $q$ :

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

Figure 4.3 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).

<sup>1</sup>All examples used in this section are real instances of co-change clusters, extracted from the subject systems used in this paper, see Section 6.1.2.





**Figure 4.3.** Crosscutting cluster (SysPol)

*Black Sheep*: whenever a cluster is spread over some packages 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:

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

Figure 4.4 shows an example of Black Sheep Cluster. The cluster red is spread over three directories and touch very few files in each one (its focus is 0.09).

We also define two co-change patterns with similar behavior. Basically, a cluster  $q$  has a body  $B$  and a set of arms  $T$ . Most source code 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, which we name Octopus Cluster:

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

Figure 4.5 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

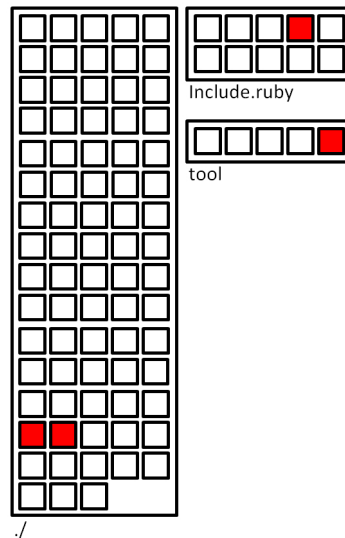


Figure 4.4. Black Sheep cluster (Ruby)

sub-cluster, this tentacle has focus 0.005. Finally, the whole Octopus has focus 0.78, which avoids its classification as Crosscutting or Black-Sheep.

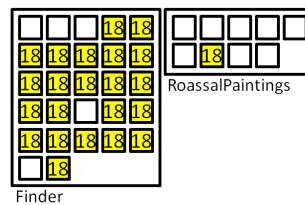
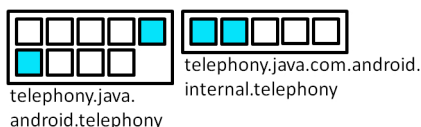


Figure 4.5. Octopus cluster (Moose)

The following thresholds are set to represent a cluster  $q$  with these properties, which we name Squid Cluster:

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

Figure 4.6 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. Finally, this cluster has focus 0.31, which avoids its categorization as Crosscutting or Black Sheep.



**Figure 4.6.** Squid cluster (Platform Frameworks)

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

As usual in the case of metric-based rules to detect code patterns [Marinescu, 2004; Lanza and Marinescu, 2006], 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 [Silva et al., 2014b, 2015a]. Typically, low focus values are smaller than 0.3 and high spread values are greater or equal to four packages.

## 4.2 ModularityCheck

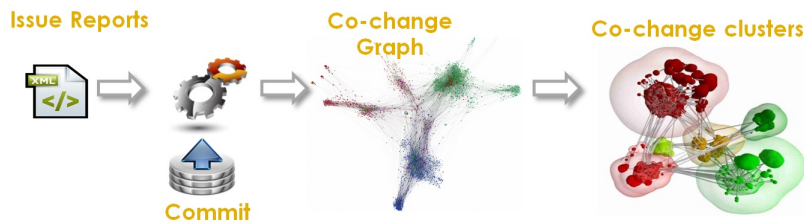
There is a growing interest in tools to enhance software quality [Kersten and Murphy, 2006; Zimmermann et al., 2005]. Specifically, several tools have been developed for improving software modularity [Rebêlo et al., 2014; Vacchi et al., 2014; Bryton and Brito e Abreu, 2008; Schwanke, 1991]. Most of such tools help architects to understand the current package decomposition. Basically, they extract information from the source code, using structural dependencies or the source code text [Robillard and Murphy, 2007, 2002].

In this section, we present ModularityCheck tool for understanding and supporting package modularity assessment using co-change clusters. The proposed tool has the following features:

- The tool extracts commits automatically from the version history of the target system and discards noisy commits by checking their issue reports.
- The tool retrieves set of classes that usually changed together in the past, which we termed co-change clusters, as described in Chapter 3.
- The tool relies on distribution maps to reason about the projection of the extracted co-change clusters in the tradition decomposition of a system in packages. It also calculates a set of metrics defined for distribution maps to support the characterization of the extracted co-change clusters.

### 4.2.1 ModularityCheck in a Nutshell

ModularityCheck supports the following stages to assess the quality of a system package modularity: pre-processing, post-processing, co-change clusters retrieval, and cluster visualization. Figure 4.7 shows the process to retrieve co-change clusters. A detailed presentation of this process is described in Section 3.1.



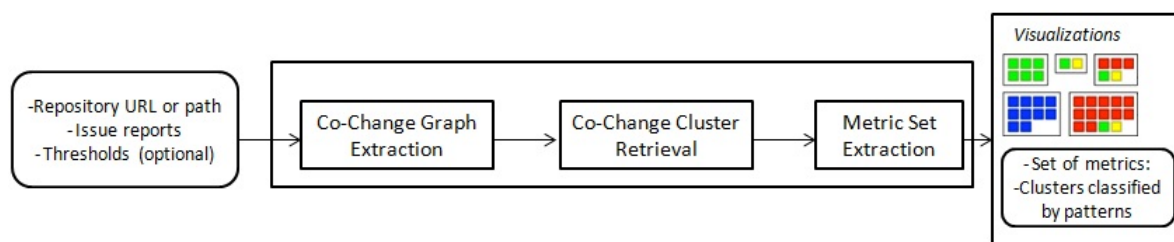
**Figure 4.7.** ModularityCheck's overview.

In the first stage, the tool applies several preprocessing tasks which are responsible for selecting commits from version history to create the co-change graph. After that, the co-change graph is automatically processed to produce a new modular facet: co-change clusters, which abstract out common changes made to a system, as stored in version control platforms. Finally, the tool uses distribution maps to reason about the projection of the extracted clusters in the traditional decomposition of a system in packages. ModularityCheck also provides a set of metrics defined for distribution maps. Particularly, it is possible to detect recurrent distribution patterns of co-change clusters listed by the tool.

### 4.2.2 Architecture

ModularityCheck supports package modularity assessment of systems implemented in the Java language. The tool relies on the following inputs: (i) the issue reports saved in XML files; (ii) URL of the version control platform (SVN or GIT). (iii) maximum number of packages to remove highly scattered commits. (iv) minimum number of classes in a co-change cluster. We discard small clusters because they may eventually generate a decomposition of the system with hundreds of clusters. Figure 4.8 shows the tool's architecture which includes the following modules:

**Co-Change Graph Extraction:** As illustrated in Figure 4.8, the tool receives the URL associated to the version control platform of the target system and the issue reports. When extracting co-change graphs, it is fundamental to preprocess the considered commits to filter out commits that may pollute the graph with noise. Firstly,



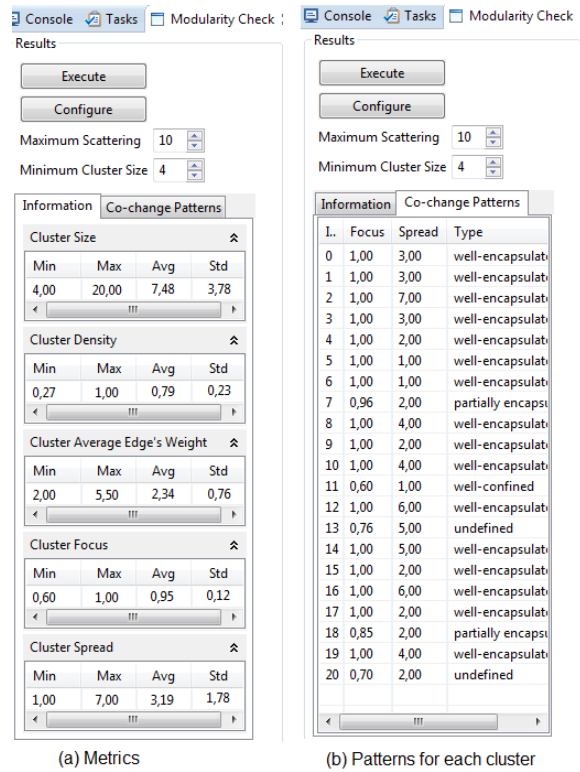
**Figure 4.8.** ModularityCheck’s architecture.

the tool removes commits not associated to maintenance issues because such commits may denote partial implementations of programming tasks. Secondly, the tool removes commits not changing classes because the co-changes considered by ModularityCheck are defined for classes. Thirdly, commits associated to multiple maintenance issues are removed. Finally, the last pruning task removes highly scattered commits, according the *Maximum Scattering* threshold, an input parameter.

**Co-Change Cluster Retrieval:** After extracting the co-change graph, a post-processing task is applied to prune edges with small weights. Then, in a further step, Chameleon is performed to retrieve subgraphs with high density. The number of clusters is defined by executing Chameleon multiple times. After each execution, small clusters are discarded according to the *Minimum Cluster Size* threshold informed by the user. The default value considered by the tool is four classes, i.e., after the clustering execution, clusters with less than four classes are removed.

**Metric Set Extraction:** The tool calculates the number of vertices, edges, and co-change graph’s density before and after the post-processing filter. After retrieving the co-change clusters, the tool presents the final number of clusters and several standard descriptive statistics measurements. These metrics describe the size and density of the extracted co-change clusters, and cluster average edges’ weight. Moreover, the tool presents metrics defined for distribution maps, like focus and spread. ModularityCheck also allows developers to inspect the distribution of the co-change clusters over the package structure by using distribution maps. In the package structure, we only consider classes that are members of co-change clusters, in order to improve the maps visualization. Finally, all classes in a co-change cluster have the same color.

After measuring focus and spread, the tool classifies recurrent distribution patterns of co-change clusters as previously presented in this chapter. In order to



**Figure 4.9.** Filters and metric results.

present ModularityCheck, we provide a scenario of usage involving information from the Geronimo Web Application Server system, extracted during 9.75 years (08/20/2003 - 06/04/2013). Figure 4.9 shows the results concerning co-change clustering. A detailed discussion of such results is presented in Section 3.

### 4.3 Final Remarks

In this chapter, we described six co-change patterns that represent common instances of co-change clusters. We relied on focus, touch, and spread metrics to characterize such patterns detected in the clusters. We also described a tool (plugin for Eclipse) that provides visualizations about co-change clusters. The ultimate goal of ModularityCheck is to extract recurrent patterns to support the comprehension and analysis of classes that usually change together. This tool can be used as an alternative view during maintenance tasks to improve developers' comprehension and detect possible design anomalies. The proposed tool and information on how to execute it is available at: <http://aserg.labsoft.dcc.ufmg.br/modularitycheck>.

# Chapter 5

## Developers' Perception on Co-change Clusters

In this chapter, we report an empirical study with experts on six systems to reveal developers' view on the usage of Co-Change Clustering for assessing modular decompositions. We start by presenting the research method and steps followed in the study (Section 5.1). Then, we report the developers' view on co-change clusters and the main findings of this study (Section 5.2). In Section 5.3, we put in perspective our findings and the lessons learned from the study. Finally, we discuss threats to validity (Section 5.4).

### 5.1 Study Design

In this section we present the questions that motivated our research (Section 5.1.1). We also present the dataset (Section 5.1.2), the threshold selection (Section 5.1.3), and the steps we followed to extract the co-change clusters (Section 5.1.4), and to conduct the interviews (Section 5.1.5).

#### 5.1.1 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.

### 5.1.2 Target Systems and Developers

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 Nierstrasz et al. [2010], 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). These systems were selected due to the availability of systems' developers to participate of the study.

Table 5.1 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.

### 5.1.3 Thresholds Selection

For this evaluation, we use the same threshold selections of our previous experience in Chapter 3.2. The thresholds are as follows:



**Table 5.1.** Target systems

<b>System</b>	<b>LOC</b>	<b>NOP</b>	<b>NOC</b>	<b>Commits</b>	<b>Period</b>
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

- $MAX\_SCATTERING = 10$  packages, i.e., we discard commits changing classes located in more than ten packages.
- $MIN\_WEIGHT = 2$  co-changes, i.e., we discard edges connecting classes with less than two co-changes.
- $M\_INITIAL = NOC_G * 0.20$ , i.e., the clustering algorithm starts with a number of partitions that is one-fifth of the number of classes in the co-change graph ( $NOC_G$ ).
- $MIN\_CLUSTER\_SZ = 4$  classes, i.e., after each clustering execution, we remove clusters with less than 4 classes.

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.

### 5.1.4 Extracting the Co-Change Clusters

We start by preprocessing the extracted commits to compute co-change graphs. Table 5.2 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 5.2.** 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 the preprocessing phase, we extracted a co-change graph for each system. Then, we applied the post-processing filters to remove edges with unitary weights and connected components smaller than 4, as defined by the *MIN\_CLUSTER\_SZ* threshold. Table 5.3 shows the number of vertices ( $|V|$ ) and the number of edges ( $|E|$ ) in each co-change graph, before and after pruning edges with weight 1 and their densities, respectively. It also presents the number of connected components before and after the post-processing task for each co-change graph.

**Table 5.3.** Number of vertices ( $|V|$ ), edges ( $|E|$ ), co-change graphs' density (D), and number of connected components before and after the post-processing filter

System	Post-Processing					
	Before			After		
	$ V $	$ E $	D	$ V $	$ E $	D
SysPol	514	35,753	0.27	505	25,613	0.20
Seaside	1,042	100,415	0.18	797	25,272	0.08
Moose	484	9,610	0.08	217	1,313	0.06
Fuel	299	4,184	0.09	64	222	0.11
Epicea	301	3,336	0.07	153	805	0.07
Glamour	659	7,616	0.03	287	1,380	0.03

As can be observed in Table 5.3, co-change graphs are sparse graphs, having low density in Pharo and 0.2 in SysPol systems after the pruning phase. Moreover, the graphs had a substantial reduction in number of edges. About 28.4% (SysPol), 74.8% (Seaside), 86.3% (Moose), 94.7% (Fuel), 75.9% (Epicea), and 81.9% (Glamour) of the edges have unitary weights, i.e., these edges connect classes that changed together just once. Finally, the vertice number after graph pos-processing was reduced to 98% (SysPol), 76.5% (Seaside), 45% (Moose), 21.4% (Fuel), 51% (Epicea), and 43,5% (Glamour).

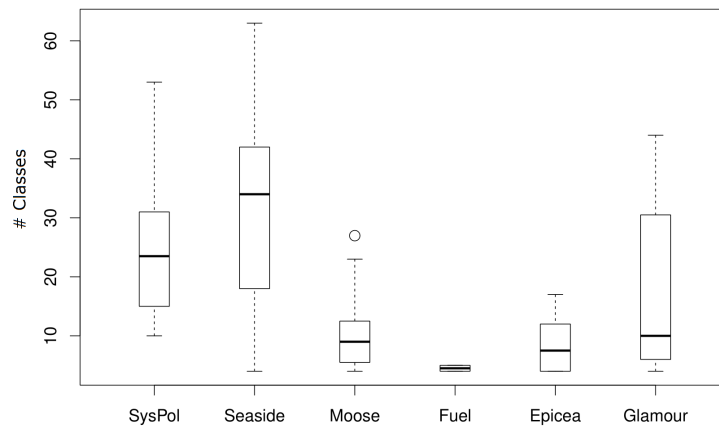
After applying the preprocessing and post-processing filters, we use the ModularityCheck tool to compute the co-change clusters [Silva et al., 2014a]. This tool relies on the Chameleon algorithm to compute co-change clusters, as described in Chapter 3. Table 5.4 shows the initial number of co-change clusters retrieved by the clustering algorithm and the number of clusters after discarding the small clusters, as defined by the *MIN\_CLUSTER\_SZ* threshold. Finally, the table also presents the ratio between the number of clusters after pruning small clusters and the number of packages (as shown in column %NOP). Table 5.4 shows the number of co-change clusters computed for each system (102 clusters, in total).

**Table 5.4.** Number of co-change clusters

System	# clusters		%NOP
	All	$ \mathbf{V}  \geq 4$	
SysPol	21	20	0.05
Seaside	30	25	0.19
Moose	24	20	0.55
Fuel	24	8	0.19
Epicea	24	16	0.19
Glamour	24	15	0.62

The initial number of clusters retrieved was 21, 30, 24, 24 and 24 for SysPol, Seaside, Moose, Fuel, Epicea, and Glamour systems, respectively. After extracting co-change clusters by Chameleon, we discarded clusters smaller than four, since they represent very small modules. As an example, for the Glamour system was selected 15 clusters, since the others have less than four classes.

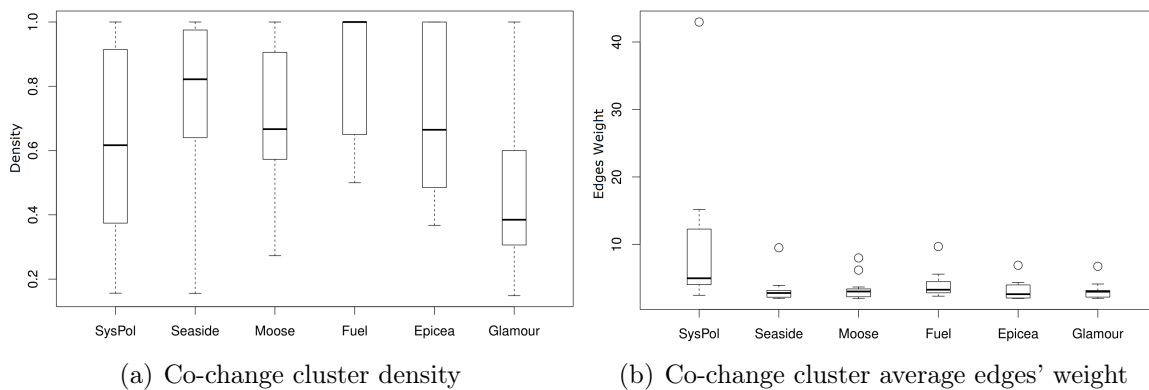
Figure 5.1 shows the distribution of cluster sizes, in terms of class number. As an example, the co-change clusters have  $25 \pm 11.5$  classes,  $31.2 \pm 17.6$  classes, and  $9 \pm 1.4$  classes (average  $\pm$  standard deviation) in the SysPol, Seaside, Epicea systems, respectively. Furthermore, the biggest cluster for SysPol (53 classes), Seaside (63 classes), and Glamour (44 classes) systems contain a significant number of classes.



**Figure 5.1.** Co-change cluster sizes

Figure 5.2-a 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 a high probability of co-changes between each pair of classes in the cluster. The clusters on SysPol have a median density of 0.62, 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).

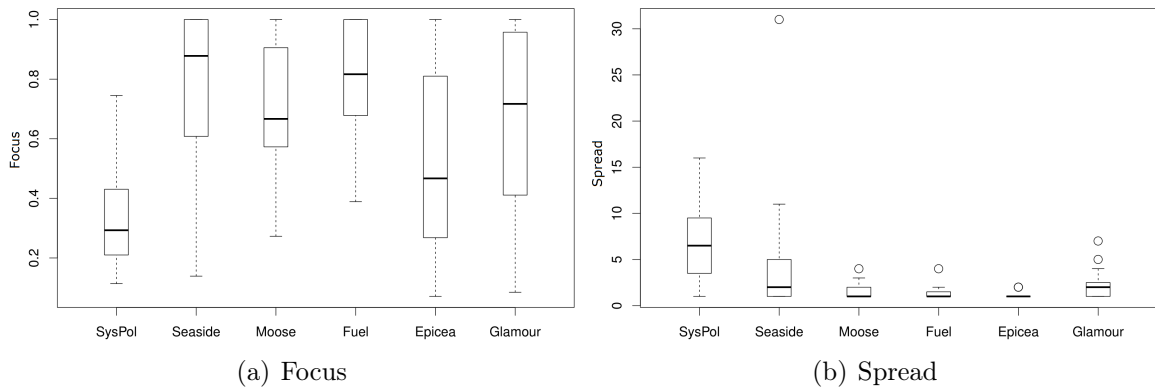
We interviewed Pharo's developers personally and the SysPol developer by Skype. In the interviews, we presented each co-change cluster that matched one of the proposed co-change patterns to the developers. For each cluster we showed to the developers its classes and in some particular cases the commits responsible by the co-changes represented by the cluster. We then asked the developers to elaborate on the concerns implemented by the clusters and on possible design anomalies evidenced by them.



(a) Co-change cluster density

(b) Co-change cluster average edges' weight

**Figure 5.2.** Distribution map metrics



**Figure 5.3.** Distribution map metrics

Figure 5.2-b shows standard descriptive statistic measurements for edge’s average weight in the extracted clusters. The average weight in a cluster is the sum of all edge’s weights divided by the number of edges. We can observe that the median edges’ weight is not high, being 5 in SysPol’s and a bit greater than 2 in Pharo’s systems. Moreover, we analyzed the edge’s weight range in several co-change graphs. As an example, for SysPol system, there are classes in a cluster that changed together massively (more than 300 times) and most class pairs changed 20 times on average (edges’ weight), displayed as an outlier in Figure 5.2-b.

Figure 5.3 show the distribution regarding focus and spread of the co-change clusters for each system. We can observe that the co-change clusters in Seaside, Moose, and Fuel have a higher focus than the others. For example, the median focus in Seaside is 0.88, against 0.29 in SysPol. Regarding spread, Moose, Fuel, and Epicea have a lower value than the others, the median is 1 against 6.5 (SysPol) and 2 for Seaside and Glamour.

### 5.1.5 Interviews

Table 5.5 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.

We conducted face-to-face and Skype semi-structured interviews with each developer using open-ended questions [Flick, 2009]. During the interviews, we presented all co-change clusters that matched one of the proposed co-change patterns to the devel-

**Table 5.5.** Expert Developers' Profile

<b>System</b>	<b>Developer ID</b>	<b>Experience (Years)</b>	<b># Emails/Chats</b>
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

opers. We used Grounded Theory [Corbin and Strauss, 1990] 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 5.5 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.

## 5.2 Results

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

### 5.2.1 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 Chapter 4.1. The results are summarized in Table 6.3. 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.

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 Chapter 4.1.

**Table 5.6.** 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%)

**Table 5.7.** 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 encoding 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

## 5.2.2 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 5.7. 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 5.7 include classes that implement clear and 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 5.8 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 5.8, 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 5.8). 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:

**Table 5.8.** Concerns implemented by Crosscutting clusters in SysPol

ID	Spread	Focus	Size	Codes
1	9	0.26	45	<u>Several</u> concerns, search case, search involved in crime, insert conduct
2	9	0.22	29	<u>Several</u> concerns, seizure of material, search for material, and create article
3	10	0.20	31	Requirement related to the concern <i>analysis</i> , including review analysis and analysis in flagrant
4	12	0.15	31	<u>Several</u> classes are associated to the <i>task</i> and <i>consolidation</i> concerns
5	15	0.29	35	Subjects related to create article and to prepare expert report
6	9	0.22	24	<u>Several</u> 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



“*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. [Kiczales et al., 1997b, 2001]. 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:** The codes extracted from developers’ answers for clusters classified as Octopus are summarized in Table 5.9. From the developers’ answers, we observed that all Octopus represent *Partially Encapsulated Concerns* (category), as illustrated by the following clusters:

- In Moose, there is an Octopus (see Figure 5.4-a) 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.
- In Glamour, there is an Octopus (see Figure 5.4-b) 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).

**Table 5.9.** Concerns implemented by Octopus clusters

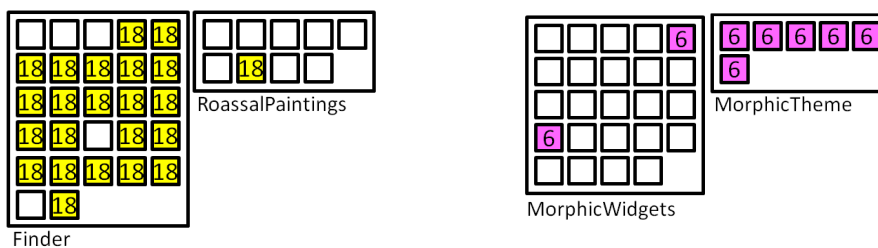
System	ID	Focus	Spread	Size	Codes
SysPol	1	0.67	6	23	Pagination of the searches
	2	0.45	8	23	Creation, insertion, and grouping of articles in a proceeding
	3	0.67	16	53	The functionality searching and validating
	4	0.75	3	19	Task validation
	5	0.55	5	13	Material functionality
Seaside	6	0.56	4	25	Tool for development applications
	7	0.90	2	18	Welcome to seaside page
	8	0.41	3	63	Core configuration of Seaside
	9	0.95	2	34	Widgets to develop applications
	10	0.83	5	57	Interface between Seaside and different dialects
	11	0.61	9	45	Developer was not able to describe this cluster
	12	0.75	11	58	Generation of XHTML markup, administration tools, sending of emails, encapsulation of parts in a page
	13	0.77	5	35	Enable Seaside runs on the Comanche HTTP and Swazoo Web server, stream events from the server to client
	14	0.45	8	42	Widget features based on AJAX, development and debugging support, rendering configuration
	15	0.88	5	38	Tool related to code browser
Moose	16	0.82	2	16	Developer was not able to describe this cluster
	17	0.55	3	27	Model that parses classes of a O.O. system
Fuel	18	0.78	2	23	Interface update of the Moose Panel
	19	0.80	4	4	Different ways of serializing samples
Epicea	20	0.81	2	17	Change sets extraction from commits and training of the classifier
	21	0.43	2	12	Persistence services
Glamour	22	0.77	2	8	Theme implementation for Glamour
	23	0.40	2	10	Browser implementation

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

We 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 5.7). 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)



(a) Moose

(b) Glamour

**Figure 5.4.** Octopus cluster

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 4.5):

*“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.

## 5.3 Discussion

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

### 5.3.1 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 [Silva et al., 2015a]. 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)

### 5.3.2 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” [Tarr et al., 1999], 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 [Evans, 2003]. We plan to investigate whether information systems architected using such principles are less subjected to crosscutting changes, as the ones we found in SysPol.

### 5.3.3 (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 [Apel and Kästner, 2009; Kästner et al., 2008], 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.

### 5.3.4 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 [Murphy-Hill et al., 2009]. To tackle this limitation, we can use tools that automatically detect refactorings from version histories, like Ref-Finder [Prete et al., 2010] and Ref-Detector [Tsantalis et al., 2013]. 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.

## 5.4 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.

## 5.5 Final Remarks

One of the benefits of modularity is managerial, by allowing separate groups to work on each module with little need for communication [Parnas, 1972]. 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 chapter, we evaluate in the field the technique proposed in this thesis 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 around 50% 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.





# Chapter 6

## A Large Scale Study on Co-Change Patterns

In this chapter, we report a large scale study using projects hosted in GitHub. We start by presenting the steps followed to collect this corpus (Section 6.1). Then, we evaluate the clustering quality by comparing with another clustering measures (Section 6.2). Section 6.3 characterizes co-change clusters and analyzes the pattern distribution concerning programming languages. In Section 6.5, we conduct a series of empirical analysis based on regression modeling and semantic analysis. Finally, we discuss threats to validity (Section 6.6).

### 6.1 Study Design

In this section we present the research questions (Section 6.1.1) and the criteria followed to select the projects used in this study (Section 6.1.2). We also present the criteria used to select the threshold used for preprocessing commits and extracting co-change clusters (Section 6.1.3).

#### 6.1.1 Research Questions

We formulate four research questions to evaluate whether the proposed co-change cluster patterns have different impacts on some software engineering output of interest, such as, the level of co-changeness, the level of activity on clusters, the number of developers working on clusters, and the level of ownership on clusters.

*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 aim in this RQ, we investigate 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 (number of classes) 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. With this RQ, we aim at investigating 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?*

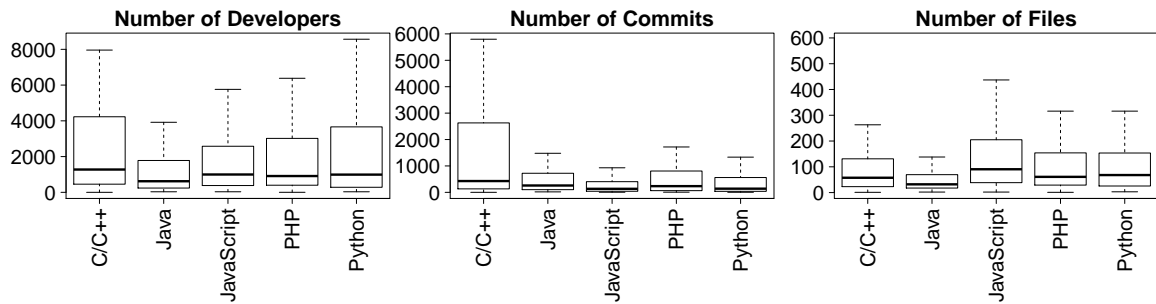
The number of developers who work on the classes of a cluster may indicate different points, e.g., the heterogeneity of the team working on a piece of the systems or the effort (manpower).

*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].

### 6.1.2 Dataset

We describe the projects in GitHub that we collected, the selection criteria to generate the corpus and the analysis methods we use to answer our research questions. We aim at conducting 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,



**Figure 6.1.** The overall number of commits and number of files by 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 6.1. 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 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 the this project.

We included 133 systems in our dataset. Table 6.1 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 375K files and 41 MLOC, respectively.

**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

**Table 6.1.** Projects in GitHub organized by language

Language	Projects	Commits	Developers	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

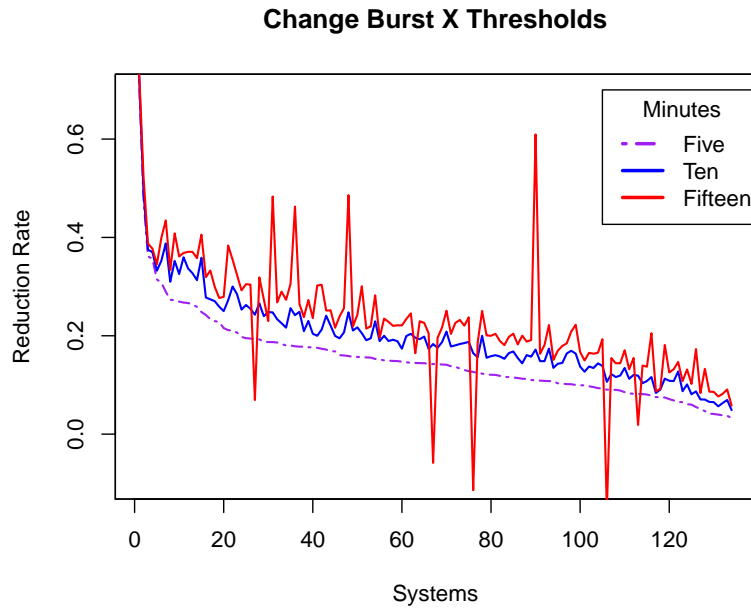
to tests, and vendored files. We used Linguist tool<sup>1</sup> to generate language breakdown graphs. Linguist is a tool used by GitHub to compute the percentage of files by programming language in a repository. We follow Linguist’s recommendations to remove files from our dataset. The tool classified automatically 129,455 files (34%), including image, xml, txt, js, php, and c 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%).

### 6.1.3 Threshold Selection

First, we preprocess the commit transactions of each system to compute co-change graphs. For this evaluation, we use the same thresholds of our previous experiences with Co-Change Clustering, as presented in Chapters 3 and 5, with the exception of 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]. Figure 6.2 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. However, for 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 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

<sup>1</sup><https://github.com/github/linguist>

only merging commits associated to the same task. For these 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 manually inspected several log messages to check whether the changes concern the same maintenance task.



**Figure 6.2.** Commit reduction rate for 5, 10, and 15 minutes

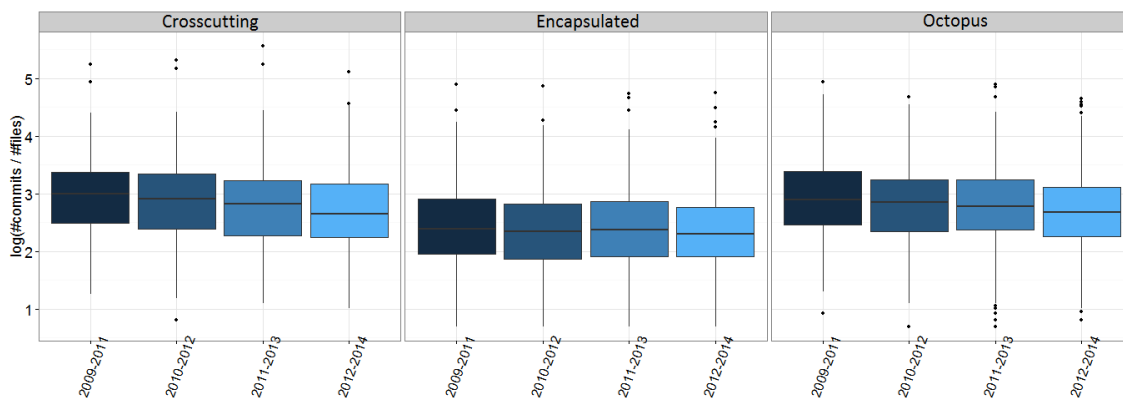
Finally, ten systems do not provide enough commits to mine co-change clusters and they are discarded, thus, we extract co-change clusters for 123 projects. Our selection includes well-known systems, such as `torvalds/linux`, `php/php-src`, `ruby/ruby`, `rails/rails`, and `webscalesql/webscalesql-5.6`.

## 6.2 Clustering Quality

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 (Encapsulated, Well-defined, Crosscutting, Black-sheep, Squid, and Octopus). Figure 6.3 shows the evolution of commit densities per co-change pattern. Note that for all patterns, there is no significant difference of commit density.



**Figure 6.3.** Evolution of commit density (source code files) per co-change pattern

### 6.2.1 Analysis of the Co-Change Cluster Extraction

This thesis proposes the function *Coefficient* (Chapter 3.1.2) 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—for all projects we have 600 time windows (clusterings). 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]. Coverage values also ranges from 0 to 1, 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 evaluate the effectiveness of co-change clusters.<sup>2</sup> 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 6.4 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.

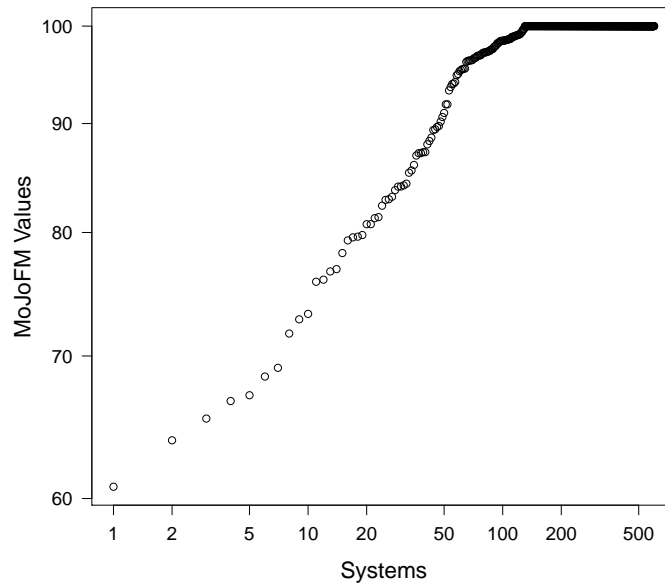
## 6.3 Classifying Co-Change Clusters

We classify the extracted co-change clusters into six patterns: Encapsulated, Well-Confined, Crosscutting, Black-Sheep, Squid, and Octopus Clusters. Table 6.2 summarizes 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. We can observe a significant difference—concerning number of categorized clusters—from the results obtained in Chapter 5. The number of co-change patterns considered in this study is the reason for the high percentage of categorized clusters.

Nonetheless, we group these clusters to ease the analysis, as follows: (i) clusters with localized changes: Encapsulated and Well-Confined Clusters, (ii) clusters which

---

<sup>2</sup>To calculate MoJoFM, we relied on the MoJo 2.0, <http://www.cs.yorku.ca/~bil/downloads/>.



**Figure 6.4.** MoJoFM values for 600 clusterings

**Table 6.2.** 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 of Co-Change Patterns		1,719 (95.39%)
Total of Extracted Clusters		1,802 (100%)

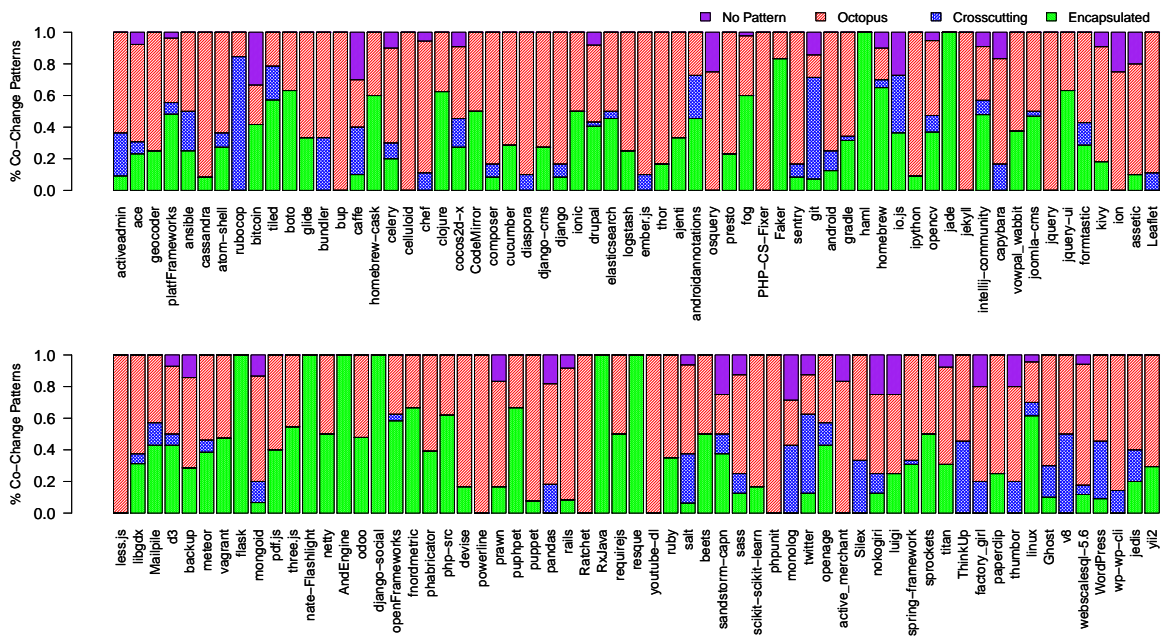
present crosscutting behavior: Crosscutting and Black-Sheep Clusters, (iii) clusters with body and arms: Squid and Octopus Clusters. Table 6.3 summarizes the co-change clusters as grouped in these three major patterns: clusters with confined changes are named Encapsulated (Encapsulated and Well-Confined), clusters with crosscutting behavior as Crosscutting (Crosscutting and Black-Sheep), and clusters with body and arms as Octopus (Squid and Octopus). As we can observe, instances of Octopus Clusters are quite common, since they are present on 114 systems (93%), representing 48.33% of all co-change clusters. Furthermore, 38.35% of the clusters are Encapsulated Clusters, which are covering 95 systems (77%). By contrast, Crosscutting Clusters are detected in only 57 systems (46%) and they represent 8.71% of the co-change clusters.

Figure 6.5 depicts the percentage of co-change clusters by system in stacked bar plots. The bars are grouped by pattern (dashed bars) or pattern absence (solid



**Table 6.3.** Number and percentage of categorized co-change clusters grouped in three major patterns

Pattern	# Systems	# Clusters
Encapsulated	95 (77%)	691 (38.35%)
Crosscutting	57 (46%)	157 (8.71%)
Octopus	114 (93%)	871 (48.33%)
No Pattern	38 (31%)	83 (4.61%)
Total Coverage of Co-Change Patterns		1,719 (95.39%)
Total of Extracted Clusters		1,802 (100%)

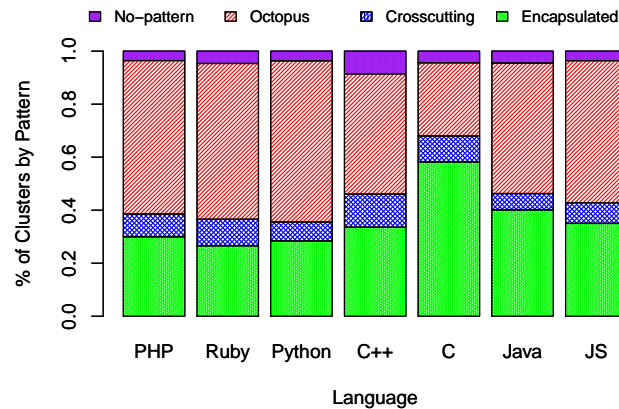
**Figure 6.5.** Relative number of identified and classified clusters for each system

purple bars). We found instances of co-change patterns in all 123 systems. Specifically, all systems have Encapsulated or Octopus Clusters. Octopus are present in most systems and the average percentage of such clusters by system is 57% (dashed red bars). The number of systems with Encapsulated Clusters is smaller than Octopus but it is significant, 30% on average (dashed green bars). By contrast, clusters that present Crosscutting behavior are relatively rare. On average, the percentage of Crosscutting is 9% (dashed blue bars).

We identify eight systems that have all co-change clusters categorized as Encapsulated (dashed green bars). Furthermore, all clusters in 10 systems are Octopus (dashed red bars). In addition, 85 projects have all co-change clusters categorized and 54 projects have only Encapsulated and/or Octopus Patterns. By contrast, Cross-

cutting Clusters do not dominate any system, e.g., 58% of the systems with clusters categorized as Crosscutting have only one cluster and 72% have two clusters. `JetBrains/intellij-community` and `torvalds/linux` are the top two systems concerning absolute number of Crosscutting Clusters, with 13 and 29 clusters, respectively. Crosscutting Clusters in `Intellij-Community` represent 9% of extracted clusters, while in `Linux` they represent only 8%.

Figure 6.6 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 Crosscutting Pattern. For instance, PHP and C++ projects have 9% and 12% of their respective clusters categorized as Crosscutting Clusters, respectively. Conversely, Octopus Pattern is very common. The percentage of Octopus Clusters is more than 50% on average, with exception for C projects (28%). Encapsulated is also common in all languages (36% on average).



**Figure 6.6.** Relative Co-Change Pattern coverage by Programming Language

## 6.4 Statistical Methods

We use regression modeling to describe the relationship of a set of predictors against a response. To answer the research questions proposed in Section 6.5, we model either the number of co-change commits or the number of commits in clusters against other factors. 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. NBR can also handle over-dispersion [Cohen et al., 2003]. For example, there are cases where the response variance is greater than the mean. For

this reason, 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 in the predictor variable, for a coefficient  $\beta_i$ , a one unit change in  $\beta_i$  yields an expected change in the response of  $e_i^\beta$ , 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.

To check whether excessive multi-collinearity is an issue, we compute the variance inflation factor (VIF) of each dependent variable in all models. 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 models for a wider audience, we also provide Fox’s effect plots [Fox, 2003]. 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.

## 6.5 Analysis of the Results

In this section, we conduct a series of statistical analysis and an investigation underlying natural language topics in commit messages.

### 6.5.1 Statistical Results

Prior to analyzing the rationale behind a co-change pattern classification, we begin with four research questions to evaluate 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. Table 6.4 shows the NBR model for number of co-change bursts per system. We include some variables as controls for factors that influence co-change bursts. For example, the number of source code files in a project is included because *size* may induce a greater number of co-change relations. Moreover, project age may also impact the analysis because older projects have substantial 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 Encapsulated, 29 Crosscutting, and 88 Octopus Clusters.

**Table 6.4.** 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	***
nEncapsulated	-3.549e-02	1.615e-02	-2.197	0.028008	*
nOctopus	1.154e-01	1.587e-02	7.271	3.58e-13	***
nCrosscutting	1.602e-01	3.523e-02	4.546	5.46e-06	***

Table 6.5 shows that all variables are significant, with the exception of Encapsulated Clusters, i.e., those factors account for some of the variance in the ripple effect. 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 *Octopus* Clusters accounts for 11% of the total explained deviance, and the number of *Crosscutting* Clusters accounts for 7% of the deviance. Therefore, these relationships—Octopus and Crosscutting—are statistically significant with an important effect.

The column Estimate in the model presented in Table 6.4 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 *nEncapsulated* variable. A significant result is that for each added *Octopus* and *Crosscutting*, 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 6.7, which presents the effects on co-change bursts for all variables included in the NBR model.

**Conclusion RQ #1:** *An increase in the number of Crosscutting and Octopus Clusters is associated with an increase in the number of co-change bursts. In contrast, the association with Encapsulated Clusters is not significant.*

*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 6.6 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 (Crosscutting, Octopus, or No Pattern). 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

**Table 6.5.** Deviance table for NBR model on the number of co-change bursts per system.

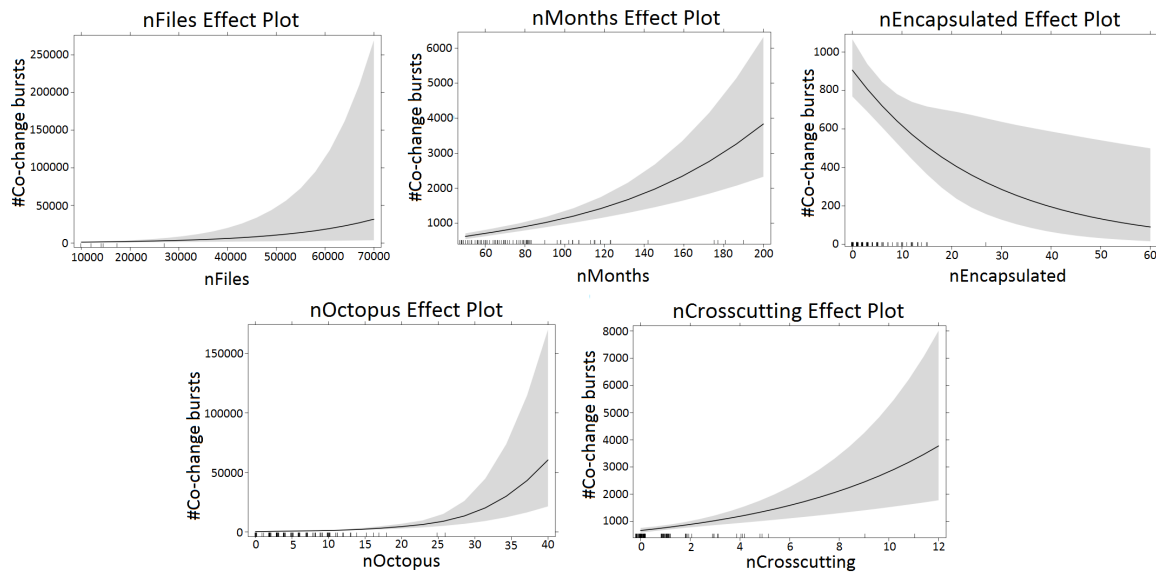
	Df	Deviance	Res. Df	Res. Dev.	Pr(>Chi)	
NULL			121	466.55		
nFiles	1	204.974	120	261.58	< 2.2e-16	***
nMonths	1	67.733	119	193.85	< 2.2e-16	***
nEncaps.	1	2.559	118	191.29	0.1097	.
nOctopus	1	37.385	117	153.90	9.698e-10	***
nCrosscut.	1	23.584	116	130.32	1.195e-06	***

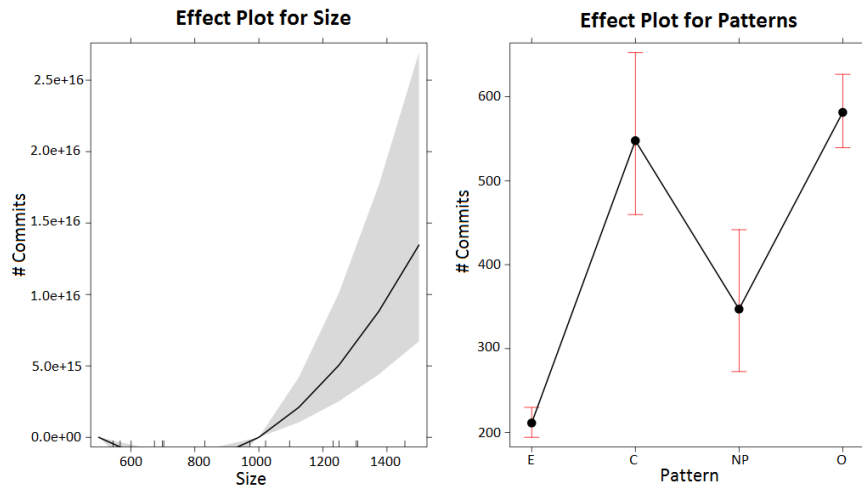
\*\*\*  $p < 0.001$ , \*\*  $p < 0.01$ , \*  $p < 0.05$ , .  $p < 0.1$

activity, we included the *size* variable in the model because some co-change patterns tend to follow a particular structure concerning their size. For instance, Octopus Clusters have more source code files than the others.

**Table 6.6.** NBR model for number of commits per cluster. C - Crosscutting Clusters, NP - Clusters with no Pattern, O - Octopus 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(C)	0.9521920	0.0989228	9.626	< 2e-16	***
factor(NP)	0.4957564	0.1300395	3.812	0.000138	***
factor(O)	1.0119010	0.0580520	17.431	< 2e-16	***

**Figure 6.7.** Effects on Co-change Bursts

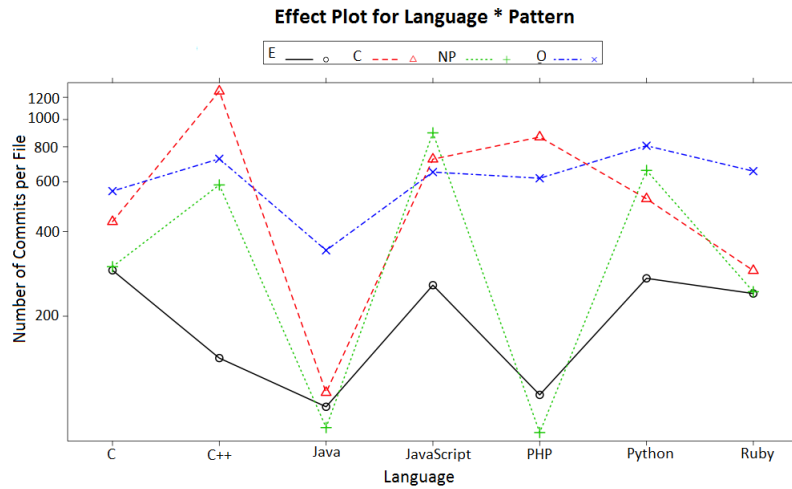


**Figure 6.8.** Effects on Activity of Clusters. E - Encapsulated, C - Crosscutting, O - Octopus, and NP - Clusters with no Pattern

The result shows that compared to the *Encapsulated* Clusters, the other clusters have higher activity level. For instance, the Octopus Clusters increase the intercept (Encapsulated Clusters) in 1.0119010 and the Crosscutting Clusters in 0.9521920. This means that the intercept for Encapsulated Clusters is  $e^{4.7181253} \approx 112$ ; for Octopus Clusters is  $e^{4.7181253+1.0119010} \approx 308$ , and for Crosscutting Clusters is  $e^{5.67} \approx 290$ . Figure 6.8 shows that *Encapsulated* Clusters have much lower level of activity than the other kind of clusters. Moreover, in Figure 6.9, we can observe a difference between the clusters activity in different programming languages. However, Encapsulated Clusters consistently have lower level of activity compared to *Octopus* and *Crosscutting* Clusters, except for Java and Ruby, where *Crosscutting* and *Encapsulated* seems to present no significant difference.

Table 6.7 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 6.6 show that Octopus and Crosscutting Clusters are statistically significant with an important effect.

**Conclusion RQ #2:** *There is a moderate and significant relationship between clusters of a given co-change pattern and the level of activity on the clusters. Octopus Clusters have a greater association with the activity level than Crosscutting Clusters, i.e., Octopus have more changes than Crosscutting. Octopus and Crosscutting Clusters have the activity level substantially higher than Encapsulated Clusters, with the*



**Figure 6.9.** Effects on Activity of Clusters. E - Encapsulated, C - Crosscutting, O - Octopus, and NP - Clusters with no Pattern

**Table 6.7.** 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 ***

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

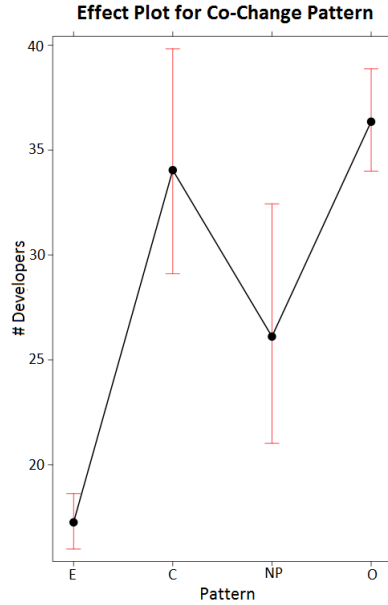
exception for Crosscutting Clusters in Java and Ruby. In other words, Encapsulated Clusters are the ones with the lowest level of changes.

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

To answer this RQ, we analyzed the number of developers who changed classes in each cluster. Table 6.8 details the NBR model for number of developers per cluster. The *size* variable is included in the model for the same reason as in RQ #2. The result shows that compared to the *Encapsulated* Clusters, the other clusters have more developers working on. For instance, Octopus Clusters increase the intercept (*Encapsulated* Clusters) in 0.7448908. This means that the intercept for Encapsulated Clusters is  $e^{2.6240082} \approx 14$  and the intercept for Octopus Clusters is  $e^{2.6240082+0.7448908} \approx 29$ . As another example, the intercept for Crosscutting Clusters is  $e^{3.303} \approx 27$ . Figure 6.10 depicts the difference among the cluster patterns concerning number of developers. *Encapsulated* Clusters usually have much less developers than clusters categorized in the other patterns.

**Table 6.8.** 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(C)	0.6793003	0.0887683	7.653	1.97e-14	***
factor(NP)	0.4141105	0.1170532	3.538	0.000403	***
factor(O)	0.7448908	0.0523115	14.240	< 2e-16	***

**Figure 6.10.** Effects on Number of Developers of Clusters. E - Encapsulated, C - Crosscutting, O - Octopus, and NP - No Pattern

Similar to RQ #2, Table 6.9 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, Octopus and Crosscutting Clusters are statistically significant with important effect.

**Conclusion RQ #3:** *There is a moderate and significant relationship between co-change pattern and number of developers per cluster. Octopus Clusters usually have more developers working on them than Crosscutting Clusters. Encapsulated Clusters are usually the ones with the lowest number of developers.*

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



**Table 6.9.** Deviance table for NBR model on the 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 ***

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

We also analyze owners of the co-change clusters per pattern. Table 6.10 details the NBR model for number of commits related to cluster's owner. We include two control variables: number of commits and developers. Number of commits provide information concerning the frequency of changes in a cluster. Additionally, number of developers in a cluster and how much they had worked on it are both important in this analysis. As we can observe in Table 6.10, in *Octopus* Clusters the clusters' owner are more important, i.e., concentrate more commits than *Encapsulated* Clusters. Interestingly, *Crosscutting* Clusters do not show a significant difference from *Encapsulated* Clusters. Octopus Clusters increase the intercept (*Encapsulated* Clusters) in 0.847 and *Crosscutting* Clusters increase the intercept only in 0.152. This means that the intercept for *Encapsulated* Clusters is  $e^{3.005} \approx 20$ ; the intercept for *Crosscutting* Clusters is  $e^{3.005+0.152} \approx 23.5$ , and the intercept for *Octopus* Clusters is  $e^{3.005+0.8472} \approx 47$ . In other words, the commits performed by the owner have higher importance in *Octopus* Clusters than in *Encapsulated* Clusters, also illustrated in Figure 6.11.

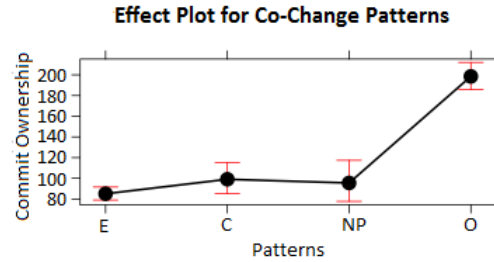
Table 6.11 shows that the variables and the *pattern* factor are significant. The number of commits in a cluster accounts for the majority of explained deviance. As we can observe, the *pattern* factor of clusters on the system accounts for 10.61% of the total explained deviance, i.e., *pattern* divided by the Deviance column.

**Conclusion RQ #4:** *There is a moderate and significant relationship between co-change patterns and commits performed by the clusters' owner. Octopus Clusters usu-*

**Table 6.10.** 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(C)	1.524e-01	8.545e-02	1.784	0.0745 .
factor(NP)	1.158e-01	1.116e-01	1.038	0.2991
factor(O)	8.472e-01	5.227e-02	16.207	<2e-16 ***

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



**Figure 6.11.** Effects on Commits of Owners. E - Encapsulated, C - Crosscutting, O - Octopus, and NP - Clusters with no Pattern

*ally have more commits related to cluster's owner than the remaining clusters. In contrast, Encapsulated and Crosscutting Clusters have no significant difference concerning number of commits.*

## 6.5.2 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.

**Table 6.11.** 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	***

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

### 6.5.2.1 Topic Extraction

*Cluster Selection Process.* For each pattern, we rank the 10 most massively changed clusters detected for 123 projects in GitHub. Table 6.12 describes the systems which contains such clusters. We apply some preprocessing 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.<sup>3</sup> 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 6.12.** The projects with the most massively changed clusters

System	Description
Ham1	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

**Encapsulated Clusters:** Table 6.13 shows the 10 most massively changed clusters classified as Encapsulated. 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 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 Ham1 cluster describes engines of templates (Ham1 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

<sup>3</sup>To extract and evaluate topics, we relied on the Mallet topic model package and guidelines, <http://mallet.cs.umass.edu/topics.php>.

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`.

**Table 6.13.** Ranking of Encapsulated Clusters

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

**Crosscutting Clusters:** Table 6.14 shows the 10 most massively changed clusters with crosscutting behavior. The table also reports the number of commits per file, cluster size, 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. As can be observed in Table 6.14, V8 system contains four crosscutting 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 `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 just one location.

**Octopus Clusters:** Table 6.15 shows the 10 most massively changed clusters classified

**Table 6.14.** Ranking of Crosscutting Clusters

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

as Octopus. The table also reports the number of commits per file, cluster size, and key words retrieved from topics to describe clusters semantically. The column *score* in the table shows the ranking of the clusters. The top one Octopus 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).

### 6.5.2.2 Timeline for Clusters

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

**Table 6.15.** Ranking of Octopus Clusters

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

**Table 6.16.** Timeline for the top one Encapsulated Cluster

Period	Encapsulated Topics	(Freq %)
2-2008	Method Add Option Util	16
1-2009	HamI docs Yard Fix Sass	34
2-2009	HamI docs Yard Fix Sass	16
1-2010	Rails Make Test	21
2-2010	HamI 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

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.

*Encapsulated Cluster - HamI project.* Table 6.16 shows the timeline of the Encapsulated 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 6.16 topics describing the cluster concern and maintenance tasks. The topic “*HamI 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*”.

**Table 6.17.** Timeline for the top one Crosscutting Cluster

Period	Crosscutting 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

*Crosscutting Cluster - V8 project.* Table 6.17 shows the timeline of the Crosscutting Cluster with the most frequent topics by semester. In this co-change cluster, the source 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.

*Octopus Cluster - 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 6.18 shows the timeline of the Octopus 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). In Chapter 5, we investigated how co-change patterns align with developers’ perception [Silva et al., 2015b]. Our results suggested that concerns implemented by Octopus Clusters tend to be very difficult to localize changes. To comprehend the findings in this Chapter, 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

**Table 6.18.** Timeline for the top one Octopus Cluster

Period	Octopus 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

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.

### 6.5.2.3 Discussion

We discuss here our findings related to the semantic analysis performed in the outlier clusters, i.e., the most changed clusters. Encapsulated Clusters seem to implement well-defined and confined concerns. The results obtained in this section reaffirm the conclusion presented in the study conducted with experts in Chapter 5. Furthermore, the most frequent maintenance activities in the top one cluster are associated to improvement tasks.

Similarly, the Crosscutting Clusters detected in projects hosted in GitHub also seem to implement well-defined concerns. In other words, they seem to implement one concern. This outcome differs from most Crosscutting Clusters obtained with experts in Chapter 5. While Crosscutting Clusters detected in the projects implement a single concern but scattered across directories (organized in different directories), SysPol’s Crosscutting Clusters implement several concerns (tangled concerns). Specifically, the interviewed experts (Chapter 5) defined the majority of clusters’ implementation as “several concerns”. In fact, the perception presented by the expert was associated to high turnover in the development team, composed by junior developers and undergraduate students. In contrast, the selected projects considered in this study were selected concerning their number of stars, a quality criterion we followed. A change made by a developer who is not a committer is only applied permanently after the committer checks and authorizes it. Nonetheless, there are Crosscutting Clusters eval-



uated in this chapter that apparently implement concerns scattered over the directory structures. Additionally, the most frequent maintenance activities in the top one Cross-cutting Cluster are related to improvement. Concerning bug fixing tasks, such problem occurred only in the two periods of the cluster’s timeline. In the later stages of the cluster’s life we could observe an absence of frequent commits associated to bugs.

Finally, we observed in the timeline of the most changed Octopus Cluster the majority of activities relies on bug fixing tasks (76% of commits). The empirical evidences found in this study concerning Octopus need further investigation. To comprehend the reason of high occurrence of bugs, we should detect whether these bugs tend to arise from changes performed between the body and tentacles or between tentacles.

## 6.6 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 <sup>4</sup>. 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 post 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 6.1.3).

## 6.7 Final Remarks

In this chapter, we conduct a large scale study in projects hosted in GitHub to evaluate the modularity assessment technique using logical coupling. We considered projects implemented in different languages, size, and domains aiming to generalize our findings. First, we measure the clustering quality using sliding time window to evaluate the

---

<sup>4</sup><http://programminghistorian.org/lessons/topic-modeling-and-mallet>.

evolution of commit density and the effectiveness of co-change clusters. We concluded that the co-change patterns remain stable overtime and most co-change clusters are well-defined sub-graphs in the co-change graph.

Concerning co-change patterns, in our investigation we observed that Octopus Clusters are indeed proportionally numerous regarding to the other patterns. Furthermore, such clusters have significant association statistically with ripple effect, activity density, diversity in the development team, and ownership.

In the semantic analysis, we concluded that Encapsulated Clusters implement well-defined and confined concerns. The most frequent maintenance activities in the top one Encapsulated Cluster are related to only improvement tasks. Similar to Encapsulated, Crosscutting Clusters evaluated in this study also implement well-defined concerns. However, these concerns are scattered over the directory structures. The most frequent maintenance activities are related to improvement and bug fixing tasks. Concerning bug fixing tasks, such problem occurred only in the two periods of the cluster's timeline. Finally, Octopus Clusters seem to implement concerns difficult to confine in packages. This result is similar to our findings in Chapter 5. We could also observe in the timeline analysis that the most common activities rely on bug fixing tasks (76% of commits).

# Chapter 7

## Conclusion

In this chapter, we summary the outcome of this thesis (Section 7.1) and review our main contributions (Section 7.2). Finally, we present further work (Section 7.3).

### 7.1 Summary

Modular decomposition is still a challenge after decades of research on new forms of software modularization. One reason is that modularization might not be viewed with single lens due to the multiple facets that a software must deal with. Research in programming languages still tries to define new modularization mechanisms that deals with such facets, such as aspects and features [Kiczales et al., 1997a; Robillard and Weigand-Warr, 2005]. In order to contribute with a solution to this problem, we proposed and evaluated a new technique to assess module decompositions. This technique relies on co-change relations to assess modularity under evolutionary (or logical) dimension. Our technique captures co-change clusters and categorizes them in co-change patterns.

After the study reported in Chapter 3, we concluded by feasibility of the technique for extracting meaningful co-change clusters using historical information. We applied Co-change Clustering to four real software systems that have approximately ten years of changes. The co-change clusters and their associated metrics were useful to assess the hierarchical modular decomposition of these systems. In Chapter 5, we reported a study with experts on six systems implemented in Java and Pharo languages to reveal the developer's perception of co-change clusters. The results show that Encapsulated Clusters are often viewed as healthy designs. Furthermore, around 50% of clusters classified as Crosscutting are associated to design anomalies. According to the interviewed developers, Octopus Clusters are usually associated to expected class

distributions which are hard to implement in an encapsulated way. Finally, Chapter 6 reports a large scale study of 123 popular software projects in GitHub. The findings reaffirm several evidences found in Chapter 5, such as the fact that Encapsulated Clusters implement well-defined concerns. Additionally, most co-change clusters are stable sub-graphs, i.e., well-defined sub-graphs (in some cases they are connected components, in others, subgraphs with few edges connecting others). Thus, the bias of the clustering measure used in this thesis is decreased because same clusters were extracted by the two different measures. Another finding from this final study is that Octopus Clusters have a significant statistical association with ripple effect, activity density, ownership, and heterogeneous development teams.

Nonetheless, we were not able to recommend modularity improvements for Crosscutting and Octopus Clusters. Seemingly, changes in a module that propagate to others—captured by Octopus Clusters—are part of planned designs, specially when it is difficult to localize concerns in a single module. Regarding Crosscutting Clusters we detected two scenarios: tangled and scattered concerns. When a Crosscutting Cluster implements tangled concerns, they tend to reveal design anomalies. Otherwise, the co-change relations seems to implement a single concern but scattered across several packages—these scatterings may represent the intended package decomposition.

## 7.2 Contributions

This thesis makes the following contributions:

- Co-change Clustering technique to systematically preprocess commits, mine co-change clusters, and assess the quality of a system’s package decomposition (Chapter 3). This technique relies on distribution maps to reason about the projection of extracted co-change clusters in the decomposition of a system in packages.
- Co-change Patterns: a set of six patterns that represent common instances of co-change clusters (Chapter 4.1). We rely on a set of metrics defined for distribution maps to characterize these patterns. After using co-change patterns to assess modularity, we were able to detect tangled, scattered, partially-encapsulated, and encapsulated concerns. Furthermore, the proposed patterns presented that most concerns in a system tend to be difficult to localize their implementation in a package (Octopus pattern). Particularly, Crosscutting clusters tend to reveal bad design.

- **ModularityCheck**: a prototype tool for extracting co-change clusters, co-change patterns, and their visual exploration using distribution maps (Section 4.2). **ModularityCheck** is publicly available on GitHub.<sup>1</sup>
- **Empirical results**: an evaluation of the proposed technique in several projects in different programming languages, with a large variety on size, age, and domains. Our findings revealed that **Encapsulated Clusters** tend to denote health modules. Furthermore, **Octopus Clusters** have significant statistical association with ripple effects (Chapter 6), but apparently they tend to represent concerns difficult to localize in modules (Chapter 5). By contrast, **Crosscutting Clusters** tend to reveal design anomalies when they implement tangled concerns. In addition, we evaluated the effectiveness of the co-change clusters and the evolution of commit densities per co-change pattern (Section 6.2). Our comparison revealed that co-change clusters are generally well-defined sub-graphs, consequently, different clustering measures can guide **Chameleon** reaches similar clusters on average. Furthermore, commit densities do not show significant difference during software evolution.

## 7.3 Further Work

We follow list possible future work on Co-change Clustering:

- An investigation on the effects that architecture styles, e.g., architectures based on domain-driven principles, have on co-change patterns. Specifically, this investigation can check whether such architectures can contribute to reduce the number of clusters matching **Crosscutting** or **Octopus** patterns. For example, in **Geronimo** we observed that the package structure is adherent to the cluster structure, i.e., localized changes usually have localized effects. We searched for **Geronimo**'s documentation and we found out that **Geronimo** uses **Inversion of Control (IoC)** techniques to decouple services and components to a high degree.<sup>2</sup>
- To compare and contrast the results of **Co-Change Clustering** with the ones generated by **Semantic Clustering** [Santos et al., 2014; Kuhn et al., 2005].
- In our experience with **Co-Change Clustering**, we found that several vertices are discarded by the algorithm. These vertices usually have high degree and they

---

<sup>1</sup><https://github.com/aserg-ufmg/ModularityCheck>.

<sup>2</sup><http://www.ibm.com/developerworks/library/os-ag-mashup-rest/> —description about architecture style used by **Geronimo**.

change with several distinct clusters. Therefore, a future work may include an investigation on whether these vertices (source code files) reveal evidences of design anomalies.

- As reported in Chapter 6, Octopus Clusters usually have a significant association with ripple effects, activity density, ownership, and heterogeneous development teams. In addition, in the semantic analysis we observed a high occurrence of commits related to bug fixing. One possible future work is to investigate the feasibility to build a bug prediction model. These models should combine logical coupling with other dimensions, such as structural and semantic relations.

# Bibliography

- Abdeen, H., Ducasse, S., and Sahraoui, H. (2011). Modularization metrics: Assessing package organization in legacy large object-oriented software. In *18th Working Conference on Reverse Engineering (WCRE)*, pages 394–398.
- Adams, B., Jiang, Z. M., and Hassan, A. E. (2010). Identifying crosscutting concerns using historical code changes. In *32nd International Conference on Software Engineering*, pages 305–314.
- Aggarwal, K. and Singh, Y. (2005). *Software Engineering*. New Age International.
- Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499.
- Alali, A., Bartman, B., Newman, C. D., and 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)*, pages 169–172.
- Almeida, H., Guedes, D., Meira, W., and 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*, pages 44–59.
- Anquetil, N., Fourrier, C., and Lethbridge, T. C. (1999). Experiments with clustering as a software remodularization method. In *6th Working Conference on Reverse Engineering (WCRE)*, pages 235–255.
- Anquetil, N. and Lethbridge, T. C. (1999). Recovering software architecture from the names of source files. *Journal of Software Maintenance*, 11(3):201–221.
- Apel, S. and Kästner, C. (2009). Virtual separation of concerns - A second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78.

- Arisholm, E., Briand, L. C., and Foyen, A. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering (TSE)*, 30(8):491–506.
- Baldwin, C. and Clark, K. (2006). *Modularity in the Design of Complex Engineering Systems*. Understanding Complex Systems.
- Baldwin, C. Y. and Clark, K. B. (2003). *Design Rules: The Power of Modularity*. MIT Press.
- Ball, T., Porter, J.-M. K. A. A., and 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., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., and De Lucia, A. (2013). An empirical study on the developers' perception of software coupling. In *International Conference on Software Engineering (ICSE)*, pages 692–701.
- Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D., and 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. and Diehl, S. (2010). Evaluating the impact of software evolution on software clustering. In *17th Working Conference on Reverse Engineering (WCRE)*, pages 99–108.
- Beyer, D. and Noack, A. (2005). Clustering software artifacts based on frequent common changes. In *13th International Workshop on Program Comprehension (IWPC)*, pages 259–268.
- Bird, C., Nagappan, N., Murphy, B., Gall, H., and 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)*, pages 4–14.
- Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *The Journal of Machine Learning Research*, 3:993–1022.
- Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008.



- Breu, S. and Zimmermann, T. (2006). Mining aspects from version history. In *21st Automated Software Engineering Conference (ASE)*, pages 221–230.
- Brin, S., Motwani, R., Ullman, J. D., and Tsur, S. (1997). Dynamic itemset counting and implication rules for market basket data. In *ACM SIGMOD International Conference on Management of Data*, pages 255–264.
- Bryton, S. and Brito e Abreu, F. (2008). Modularity-oriented refactoring. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 294–297.
- Chang, J. and Blei, D. M. (2010). Hierarchical relational models for document networks. *The Annals of Applied Statistics*, 4(1):124–150.
- Chidamber, S. and Kemerer, C. (1991). Towards a metrics suite for object oriented design. In *6th Object-oriented programming systems, languages, and applications Conference (OOPSLA)*, pages 197–211.
- Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.
- Cohen, J., Cohen, P., West, S. G., and Aiken, L. S. (2003). *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Lawrence Erlbaum.
- Corbin, J. and Strauss, A. (1990). Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21.
- Couto, C., Pires, P., Valente, M. T., Bigonha, R., and Anquetil, N. (2014). Predicting software defects with causality tests. *Journal of Systems and Software*, pages 1–38.
- Couto, C., Silva, C., Valente, M. T., Bigonha, R., and Anquetil, N. (2012). Uncovering causal relationships between software metrics and bugs. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 223–232.
- Curtis, B., Sheppard, S., Milliman, P., Borst, M., and Love, T. (1979). Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. *IEEE Transactions on Software Engineering*, SE-5(2):96–104.
- D’Ambros, M., Lanza, M., and 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., and Robbes, R. (2009b). On the relationship between change coupling and software defects. In *16th Working Conference on Reverse Engineering (WCRE)*, pages 135–144.
- D'Ambros, M., Lanza, M., and Robbes, R. (2010). An extensive comparison of bug prediction approaches. In *7th Working Conference on Mining Software Repositories (MSR)*, pages 31–41.
- Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407.
- DeRemer, F. and Kron, H. (1975). Programming-in-the large versus programming-in-the-small. In *International Conference on Reliable Software*, pages 114–121. ACM.
- Dias, M., Bacchelli, A., Gousios, G., Cassou, D., and Ducasse, S. (2015). Untangling fine-grained code changes. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 341–350.
- Dijkstra, E. W. (1974). *Selected Writings on Computing: A Personal Perspective*,. Springer-Verlag.
- Dit, B., Holtzhauer, A., Poshyvanyk, D., and Kagdi, H. (2013). A dataset from change history to support evaluation of software maintenance tasks. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 131–134.
- Ducasse, S., Gırba, T., and Kuhn, A. (2006). Distribution map. In *22nd IEEE International Conference on Software Maintenance (ICSM)*, pages 203–212.
- Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Prentice Hall.
- Flick, U. (2009). *An Introduction to Qualitative Research*. SAGE.
- Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. (2011). Jdeodorant: identification and application of extract class refactorings. In *33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039.
- Fox, J. (2003). Effect displays in r for generalised linear models. *Journal of Statistical Software*, 8(15):1–27.

- Gethers, M., Kagdi, H., Dit, B., and Poshyvanyk, D. (2011). An adaptive approach to impact analysis from change requests to source code. In *26th Automated Software Engineering Conference (ASE)*, pages 540–543.
- Griffiths, T. and Steyvers, M. (2002). A probabilistic approach to semantic representation. In *24th annual Conference of the Cognitive Science Society*, pages 381–386.
- Griswold, W. G., Yuan, J. J., and Kato, Y. (2001). Exploiting the map metaphor in a tool for software evolution. In *23rd International Conference on Software Engineering (ICSE)*, pages 265–274.
- Han, J., Kamber, M., and Pei, J. (2011). *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, third edition.
- Haney, F. M. (1972). Module connection analysis: A tool for scheduling software debugging activities. In *Fall Joint Computer Conference, Part I (AFIPS)*, pages 173–179.
- Herzig, K. and Zeller, A. (2013). The impact of tangled code changes. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130.
- Herzig, K. and Zeller, A. (Unpublished manuscript, Sep. 2011). Untangling changes.
- Hora, A., Anquetil, N., Ducasse, S., Bhatti, M., Couto, C., Valente, M. T., and Martins, J. (2012). Bug maps: A tool for the visual exploration and analysis of bugs. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 523–526.
- Janzen, D. and Volder, K. D. (2003). Navigating and querying code without getting lost. In *2nd International Conference on Aspect-oriented Software Development (AOSD)*, pages 178–187.
- Kagdi, H., Gethers, M., and Poshyvanyk, D. (2013). Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering (EMSE)*, 18(5):933–969.
- Kagdi, H. and Maletic, J. I. (2006). Mining for co-changes in the context of web localization. In *8th IEEE International Symposium on Web Site Evolution (WSE)*, pages 50–57.
- Karypis, G., Han, E.-H. S., and Kumar, V. (1999). Chameleon: hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75.

- Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in software product lines. In *30th International Conference on Software Engineering (ICSE)*, pages 311–320.
- Kawrykow, D. and Robillard, M. P. (2011). Non-essential changes in version histories. In *33rd International Conference on Software Engineering (ICSE)*, pages 351–360.
- Kersten, M. and Murphy, G. C. (2006). Using task context to improve programmer productivity. In *14th International Symposium on Foundations of Software Engineering (FSE)*, pages 1–11.
- Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 327–355. Springer Verlag.
- Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997a). Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag.
- Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997b). Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag.
- Kothari, J., Denton, T., Shokoufandeh, A., Mancoridis, S., and Hassan, A. (2006). Studying the evolution of software systems using change clusters. In *14th IEEE International Conference on Program Comprehension (ICPC)*, pages 46–55.
- Kouroshfar, E. (2013). Studying the effect of co-change dispersion on software quality. In *35th International Conference on Software Engineering (ICSE)*, pages 1450–1452.
- Kuhn, A., Ducasse, S., and Girba, T. (2005). Enriching reverse engineering with semantic clustering. In *12th Working Conference on Reverse Engineering (WCRE)*, pages 133–142.
- Kuhn, A., Ducasse, S., and Girba, T. (2007). Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243.
- Lanza, M. and Marinescu, R. (2006). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer.

- Laplante, P. A. (2003). *Software Engineering for Image Processing Systems*. CRC Press, Inc.
- Lehman, M. M. (1984). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221.
- Lent, B., Swami, A. N., and Widom, J. (1997). Clustering association rules. In *13th International Conference on Data Engineering (ICDE)*, pages 220–231.
- MacQueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297.
- Mahmoud, A. and Niu, N. (2013). Evaluating software clustering algorithms in the context of program comprehension. In *21st International Conference on Program Comprehension (ICPC)*, pages 162–171.
- Maletic, J. and 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*, pages 46–53.
- Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
- Maqbool, O. and Babri, H. (2007). Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780.
- Marinescu, R. (2004). Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM)*, pages 350–359.
- Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education.
- Mens, T., Mens, K., and Tourwé, T. (2004). Aspect-oriented software evolution. *ERCIM News*, (58):36–37.
- Meyer, B. (2000). *Object-Oriented Software Construction*. Prentice-Hall.
- Mileva, Y. M. and Zeller, A. (2011). Assessing modularity via usage changes. In *10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE)*, pages 37–40.

- Misra, J., Annervaz, K. M., Kaulgud, V., Sengupta, S., and Titus, G. (2012). Software clustering: Unifying syntactic and semantic features. In *19th Working Conference on Reverse Engineering (WCRE)*, pages 113–122.
- Mitchell, B. S. and Mancoridis, S. (2006). On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208.
- Moha, N., Guéhéneuc, Y.-G., Duchien, L., and Meur, A.-F. L. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.
- Murphy-Hill, E., Parnin, C., and Black, A. P. (2009). How we refactor, and how we know it. In *31st International Conference on Software Engineering (ICSE)*, pages 287–297.
- Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., and Murphy, B. (2010). Change bursts as defect predictors. In *21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 309–318.
- Negara, S., Vakilian, M., Chen, N., Johnson, R. E., and Dig, D. (2012). Is it dangerous to use version control histories to study source code evolution? In *26th European conference on Object-Oriented Programming (ECOOP)*, pages 79–103.
- Nierstrasz, O., Ducasse, S., and Pollet, D. (2010). *Pharo by Example*. Square Bracket Associates.
- Oliva, G. A., Santana, F. W., Gerosa, M. A., and 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)*, pages 31–40.
- Oliva, G. A., Steinmacher, I., Wiese, I., and Gerosa, M. A. (2013). What can commit metadata tell us about design degradation? In *International Workshop on Principles of Software Evolution (IWPSE)*, pages 18–27.
- Omicinski, E. (2003). Alternative interest measures for mining associations in databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):57–69.
- Ostermann, K., Mezini, M., and Bockisch, C. (2005). Expressive pointcuts for increased modularity. In *19th European Conference on Object-Oriented Programming (ECOOP)*, pages 214–240.

- Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., de Lucia, A., and Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 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.
- Parnas, D. L. (1994). Software aging. In *16th International Conference on Software Engineering (ICSE)*, pages 279–287.
- Piatetsky-Shapiro, G. (1991). Discovery, analysis and presentation of strong rules. In *Knowledge Discovery in Databases*, pages 229–248.
- Poshyvanyk, D. and Marcus, A. (2007). Using information retrieval to support design of incremental change of software. In *22th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 563–566.
- Poshyvanyk, D. and Marcus, A. (2008). Measuring the semantic similarity of comments in bug reports. In *1st International ICPC2008 Workshop on Semantic Technologies in System Maintenance (STSM)*, pages 265–280.
- Prete, K., Rachatasumrit, N., Sudan, N., and Kim, M. (2010). Template-based reconstruction of complex refactorings. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10.
- Ratiu, D., Marinescu, R., and Jürjens, J. (2009). The logical modularity of programs. In *16th Working Conference on Reverse Engineering (WCRE)*, pages 123–127.
- Rebêlo, H., Leavens, G. T., Bagherzadeh, M., Rajan, H., Lima, R., Zimmerman, D. M., Cornélio, M., and Thüm, T. (2014). Modularizing crosscutting contracts with aspectjml. In *13th International Conference on Modularity (MODULARITY)*, pages 21–24.
- Robillard, M. P. and 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.
- Robillard, M. P. and Murphy, G. C. (2002). Concern graphs: finding and describing concerns using structural program dependencies. In *24th International Conference on Software Engineering (ICSE)*, pages 406–416.

- Robillard, M. P. and Murphy, G. C. (2007). Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):1–38.
- Robillard, M. P. and Weigand-Warr, F. (2005). Concernmapper: simple view-based separation of scattered concerns. In *OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '05, pages 65–69.
- Rosen-Zvi, M., Griffiths, T., Steyvers, M., and Smyth, P. (2004). The author-topic model for authors and documents. In *20th Conference on Uncertainty in Artificial Intelligence*, pages 487–494.
- Salton, G., Wong, A., and Yang, C. S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620.
- Santos, G., Valente, M. T., and Anquetil, N. (2014). Remodularization analysis using semantic clustering. In *1st CSMR-WCRE Software Evolution Week*, pages 224–233.
- Sarkar, S., Ramachandran, S., Kumar, G., Iyengar, M., Rangarajan, K., and Sivagnanam, S. (2009). Modularization of a large-scale business application: A case study. *IEEE Software*, 26(2):28–35.
- Schwanke, R. (1991). An intelligent tool for re-engineering software modularity. In *13th International Conference on Software Engineering (ICSE)*, pages 83–92.
- Schwanke, R., Xiao, L., and Cai, Y. (2013). Measuring architecture quality by structure plus history analysis. In *35th International Conference on Software Engineering (ICSE)*, pages 891–900.
- Seacord, R. C., Plakosh, D., and Lewis, G. A. (2003). *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley.
- Silva, L. L., Felix, D., Valente, M. T., and Maia, M. (2014a). ModularityCheck: A tool for assessing modularity using co-change clusters. In *5th Brazilian Conference on Software: Theory and Practice*, pages 1–8.
- Silva, L. L., Valente, M. T., and Maia, M. (2014b). Assessing modularity using co-change clusters. In *13th International Conference on Modularity*, pages 49–60.
- Silva, L. L., Valente, M. T., and Maia, M. (2014c). Modularitycheck: A tool for assessing modularity using co-change clusters. In *5th Brazilian Conference on Software: Theory and Practice (CBSOFT), Tools Demonstration Track*, pages 1–6.



- Silva, L. L., Valente, M. T., and Maia, M. (2015a). Co-change clusters: Extraction and application on assessing software modularity. In *Transactions on Aspect-Oriented Software Development XII*, volume 8989 of *Lecture Notes in Computer Science*, pages 96–131. Springer.
- Silva, L. L., Valente, M. T., Maia, M., and Anquetil, N. (2015b). Developers' perception of co-change patterns: An empirical study. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 21–30.
- Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? In *2nd Working Conference on Mining Software Repositories (MSR)*, pages 1–5.
- Sneath, P. H. A. and Sokal, R. R. (1973). *Numerical taxonomy: The principles and practice of numerical classification*. W.H. Freeman.
- Stevens, W. P., Myers, G. J., and Constantine, L. L. (1974). Structured design. *IBM Systems Journal*, 13(2):115–139.
- Tan, P.-N., Steinbach, M., and Kumar, V. (2006). *Introduction to Data Mining*. Addison-Wesley.
- Tao, Y., Dang, Y., Xie, T., Zhang, D., and Kim, S. (2012). How do software engineers understand code changes?: An exploratory study in industry. In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 51:1–51:11.
- Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S. M. (1999). N degrees of separation: Multi-dimensional separation of concerns. In *21st International Conference on Software Engineering (ICSE)*, pages 107–119.
- Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). Qualitas corpus: A curated collection of Java code for empirical studies. In *Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345.
- Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. S. (2013). Qualitas.class corpus: A compiled version of the qualitas corpus. *Software Engineering Notes*, pages 1–4.
- Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367.
- Tsantalis, N., Guana, V., Stroulia, E., and Hindle, A. (2013). A multidimensional empirical study on refactoring activity. In *Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 132–146.

- Vacchi, E., Olivares, D. M., Shaqiri, A., and Cazzola, W. (2014). Neverlang 2: A framework for modular language implementation. In *13th International Conference on Modularity*, pages 29–32.
- Vanya, A., Hofland, L., Klusener, S., van de Laar, P., and van Vliet, H. (2008). Assessing software archives with evolutionary clusters. In *16th IEEE International Conference on Program Comprehension (ICPC)*, pages 192–201.
- Walker, R. J., Rawal, S., and Sillito, J. (2012). Do crosscutting concerns cause modularity problems? In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 1–11.
- Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., and Zhang, D. (2013). Mining succinct and high-coverage api usage patterns from source code. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 319–328.
- Wen, Z. and Tzerpos, V. (2004). An effectiveness measure for software clustering algorithms. In *12th IEEE International Workshop on Program Comprehension*, pages 194–203.
- Wong, S., Cai, Y., Kim, M., and Dalton, M. (2011). Detecting software modularity violations. In *33rd International Conference on Software Engineering (ICSE)*, pages 411–420.
- Wu, J., Hassan, A. E., and Holt, R. C. (2005). Comparison of clustering algorithms in the context of software evolution. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 525–535.
- Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009). Mapo: Mining and recommending api usage patterns. In *23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 318–343.
- Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for Eclipse. In *3rd International Workshop on Predictor Models in Software Engineering*, page 9.
- Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.