Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Exploring the Use of Co-Change Clusters in Software Comprehension Tasks

Marcos César de Oliveira

Dissertação apresentada como requisito parcial para conclusão do Mestrado Profissional em Computação Aplicada

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Coorientador
Prof. Dr. Guilherme Novaes Ramos

Brasília
2015

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

Oliveira, Marcos César de
O48e Exploring the Use of Co-Change Clusters in Software Comprehension Tasks / Marcos César de Oliveira; orientador Rodrigo Bonifácio de Almeida; co-orientador Guilherme Novaes Ramos. -- Brasília, 2015.
89 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2015.

1. Desenvolvimento de Software Orientado a Features. 2. Engenharia reversa. 3. Mineração de repositórios de software. I. Almeida, Rodrigo Bonifácio de, orient. II. Ramos, Guilherme Novaes, co-orient. III. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Exploring the Use of Co-Change Clusters in Software Comprehension Tasks

Marcos César de Oliveira

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
Departamento de Ciência da Computação/UnB

Prof.ª Dr.ª Genaína Nunes Rodrigues
Departamento de Ciência da Computação/UnB

Prof. Dr. Uirá Kulesza
Centro de Ciências Exatas/UFRN

Prof. Dr. Marcelo Ladeira
Coordenador do Programa de Pós-graduação em Computação Aplicada

Brasília, 03 de setembro de 2015

To Paula

I guarantee you, every astronaut, when he comes back from space goes up to a girl and goes, *"So did you see me up there?"*

Seinfeld, SE04 EP23 – The Pilot

Agradecimentos

A minha esposa, Paula, e filhos, Beatriz, Bruno, Gabriela e Vítor, não só pela paciência e compreensão nas intermináveis horas de dedicação e ausência que foram necessárias durante a execução desse trabalho. Mas principalmente pelo incentivo e apoio da minha esposa, sempre me motivando a tentar ir além. E também especialmente aos meus filhos pelo incentivo de poder ser uma inspiração para eles.

Ao meu orientador, professor Rodrigo Bonifácio, que demonstrou grande espírito colaborativo, sempre disposto a ajudar mesmo quando os prazos eram exíguos. Além disso, por sua grande inteligência e intuição que foram fonte de inspiração para mim. Ao meu co-orientador, professor Guilherme Ramos, por suas contribuições decisivas, que enriqueceram o resultado final. Ao Márcio Ribeiro, que se dispôs a dedicar seu valioso tempo, capacidade e experiência, às importantes contribuições ao nosso trabalho.

Aos demais professores do curso de Mestrado Profissional em Computação Aplicada, que tive a honra de conhecer e deles receber os mais valiosos ensinamentos que foram cruciais não só para o avanço desse trabalho, mas também que serão úteis em outros momentos da minha vida acadêmica e profissional. E aos colegas de mestrado, que, como ninguém, sabem como é difícil conciliar família, trabalho e estudo, e além disso buscar a excelência em tudo o que fazem.

À Secretaria de Orçamento Federal (SOF), do Ministério do Planejamento, Orçamento e Gestão, especialmente ao Coordenador-Geral de Tecnologia da Informação, Robson Rung, por me apoiar desde o início, pela confiança em minha capacidade de realizar essa tarefa, e por me dar condições de dedicar tempo a esse trabalho. E também a todos da equipe da SOF que sempre foram solidários e que são motivo de orgulho para mim por poder fazer parte desse grupo de extraordinários profissionais de tão renomada instituição. Espero poder retribuir aplicando o que aprendi e, principalmente, procurando ser um profissional melhor para a instituição e para o Brasil.

Resumo

O desenvolvimento de software orientado a características (FOSD) é um paradigma que pode ser usado, entre outros, para estruturar um sistema de software em torno de características que podem representar pequenas funcionalidades do software bem como requisitos não funcionais. Além do seu papel na estruturação do software, o uso de FOSD habilita a ativação e desativação de *features* individuais em uma dada configuração de software. Essa vantagem pode ser útil em cenários onde a variabilidade do *software* é necessária. Por outro lado, a adoção da abordagem FOSD pode ser feita em um sistema de *software* existente, torna-se necessária a aplicação de alguma técnica de engenharia reversa para extração de *features* de uma base de código legada, bem como o mapeamento dessas features para suas implementações. Essa dissertação apresenta uma nova abordagem para auxiliar nessa atividade de engenharia reversa, a qual relaciona dados históricos extraídos de *sistemas de controle de tarefas* de desenvolvimento e de mudanças em código-fonte. A abordagem se baseia em técnicas de *Mineração de Repositórios de Sofware* (MSR), especificamente o agrupamento baseado em dependências evolucionárias entre elementos do código-fonte, que leva ao descobrimento de *grupos de co-mudança.* Assim, o objetivo deste trabalho é descobrir as propriedades dos grupos de co-mudança que podem ser úteis no processo de extração de *features.* Especificamente, um conjunto de termos, associados com os grupos, que revelam conceitos que podem ajudar a identificar *features.* De acordo com os resultados obtidos, os grupos de co-mudança não possuem vantagem quando usados como unidades de modularização, mas podem revelar novas dependências que são ocultas ao desenvolvedor. Também mostram que os grupos de co-mudança possuem coesão conceitual, e que podem ser usados para extrair conceitos e termos associados com eles. Por fim, os conceitos extraídos dos grupos de co-mudança podem ser usados para construir um mapeamento entre eles e o código-fonte, e que podem ser usados como uma lista de *sementes* de entrada para métodos de expansão de *features.*

Palavras-chave: Desenvolvimento de Software Orientado a *features*, engenharia reversa, mineração de repositórios de software

Abstract

Feature-oriented software development (FOSD) is a paradigm that can be used, among others, to structure a software system around the feature concept that can represents small functionalities and non-functional requirements. Besides their role in software structure, FOSD enables the activation and deactivation of individual features in a given configuration of the software. This advantage can be useful in scenarios where the variability of the software is required. On the other hand, the adoption of FOSD can be done for an existing software system, thus, becomes necessary to apply some reverse engineering technique to extract features from a legacy code base, and also the mapping between these features and their implementations. This dissertation presents a new approach to aid in the reverse engineering activity, that relates historical data from issue tracking systems and source-code changes. The approach relies upon Mining Software Repositories (MSR) techniques, specifically the clustering based on co-evolutionary dependencies between source-code elements, which leads to the discover of co-change clusters. Thus, the goal of this work is to discover the properties of the co-change clusters that can be useful in a feature extraction process. Specifically, a set of terms, associated with the clusters, which reveal concepts that can help to identify features. According to the study results, co-change clusters have no advantage when used as a modular unit, but can reveal new dependencies that is hidden to the developer. They also show that the co-change clusters have conceptual cohesion, and can be used to extract concepts and the terms associated with them. In the end, the concepts extracted from co-change clusters can be used to build a mapping from them and the source-code, and that can be used as a input seed list to feature expansion methods.

Keywords: feature oriented software development, reverse engineering, mining software repositories

Contents

1 Introduction **1**

1.1 Purpose of this dissertation . . . 3

1.2 Rationale . . . 3

1.3 Outline . . . 3

2 Literature Review **5**

2.1 Modularity . . . 5

2.1.1 Design Structure Matrix . . . 7

2.2 Feature Oriented Software Development . . . 9

2.3 Mining Software Repositories . . . 12

2.4 Software Clustering . . . 13

2.5 Mining Source-Code Change History . . . 14

3 Unveiling and Reasoning about Hidden Dependencies Induced by Co-Evolution **16**

3.1 Chapter Abstract . . . 16

3.2 Introduction . . . 16

3.3 Background . . . 18

3.3.1 Design Structure Matrix (DSM) . . . 19

3.3.2 Architectural Metrics . . . 19

3.3.3 Clustered Cost . . . 21

3.4 Methodology . . . 22

3.4.1 Extracting Fine-Grained Version History . . . 22

3.4.2 Extracting Co-Change Clusters . . . 23

3.4.3 Extracting Static Dependencies . . . 26

3.4.4 Building DSMs . . . 26

3.4.5 Computing Metrics . . . 27

3.5 Study Settings . . . 27

3.5.1 Target Systems . . . 27

3.5.2 Selection of the Threshold Combination 27
3.6 Results 29
3.6.1 Exploratory analysis of the impact of commits and issues into fine grained entities 29
3.6.2 To what extent do the hidden dependencies induced by the co-evolution of components impact the architecture? 30
3.6.3 Is it worth to restructure the architecture of a system based on the co-evolution clusters? 35
3.7 Discussion 36
3.8 Threats to Validity 37
3.9 Related Work 38
3.9.1 Version History and Modularity 38
3.9.2 DSM and Modularity 38
3.9.3 Clustering and Remodularization 38
3.9.4 Co-change clusters and Remodularization 39
3.9.5 Differences from previous works 39
3.10 Conclusion 39

4 On the Conceptual Cohesion of Co-Change Clusters 41
4.1 Chapter Abstract 41
4.2 Introduction 41
4.3 Methodology 43
4.3.1 Extracting Fine-Grained Version History 44
4.3.2 Extracting Co-Change Clusters 44
4.3.3 Building the Similarity Index 47
4.3.4 Computing Conceptual Cohesion Metrics 48
4.4 Settings 50
4.4.1 Target Systems 50
4.4.2 Selection of the Threshold Combination 50
4.5 Results 52
4.6 Terms Extraction 58
4.6.1 First Strategy: Terms Frequency 58
4.6.2 Second Strategy: LSI 59
4.6.3 Results 60
4.6.4 Discussion 61
4.7 Implications of our results 62
4.8 Threats to Validity 62
4.9 Related Work 63

4.10 Conclusion 65

5 Conclusion **66**

5.1 Summary of the Contributions 66

5.2 Impact on the Organization 67

5.3 Future Work 68

5.3.1 Providing Seeds for Feature Expansion 68

5.3.2 Feature Location 68

5.3.3 Remodularization 69

Bibliography **70**

List of Figures

2.1 Precedence Matrix (adapted from [72].) 8
2.2 Precedence Matrix with Partitions (adapted from [72].) 8
2.3 Feature model for a subsystem belonging to an important Brazilian Financial System 10

3.1 Example of DSM 19
3.2 Example of a transitive closure 20
3.3 Metrics Extraction Process (numbered circles represent the steps.) 23
3.4 Example of co-change cluster extraction (the edges' labels specify *support count* and *confidence* respectively.) 24
- (a) Current source-code 24
- (b) Fine-grained commits from HR 24
- (c) Co-change graph (MDG) 24
- (d) Pruned co-change graph using *minimum support* equals 2 and *minimum confidence* equals 0.5 24
- (e) Co-change clusters 24

3.5 Characterization of the systems with respect to the impact of the commits and issues into the fine grained entities. 30
3.6 SIOP DSMs 31
- (a) Static 31
- (b) Static and Evolutionary 31

3.7 Derby DSMs 31
- (a) Static 31
- (b) Static and Evolutionary 31

3.8 Hadoop DSMs 31
- (a) Static 31
- (b) Static and Evolutionary 31

3.9 Eclipse UI DSMs 32
- (a) Static 32
- (b) Static and Evolutive 32

3.10 JDT DSMs . . . 32
(a) Static . . . 32
(b) Static and Evolutive . . . 32
3.11 Geronimo DSMs . . . 32
(a) Static . . . 32
(b) Static and Evolutionary . . . 32
3.12 Lucene DSMs . . . 33
(a) Static . . . 33
(b) Static and Evolutionary . . . 33

4.1 Metrics Extraction Process. The numbered circles are the activities, which are executed in order. . . . 43
4.2 Example of co-change cluster extraction (the edges' labels specify *support count* and *confidence* respectively.) . . . 46
(a) Current source-code . . . 46
(b) Fine-grained commits . . . 46
(c) Co-change graph . . . 46
(d) Pruned co-change graph, using *minimum support* equals 2 and *minimum confidence* equals 0.5 . . . 46
(e) Co-change clusters . . . 46
4.3 Sample similarity indexes computed using LSI. . . . 50
(a) Coarse-grained index . . . 50
(b) Fine-grained index . . . 50
4.4 Target System's Conceptual Cohesion. . . . 53
(a) Coarse-grained modules cohesion . . . 53
(b) Fine-grained modules cohesion . . . 53
4.5 Proportion of entities in relation to modules. . . . 55
(a) Coarse-grained modules . . . 55
(b) Fine-grained modules . . . 55
4.6 Average of commits per entity. X axes represent the last two years of change history for each system, from past (left) to present (right). . . . 57

List of Tables

2.1 Definitions for the term *feature* found in literature 9

3.1 Basic metrics about target systems. #C means number of classes and interfaces; #F, methods, attributes, and constructors; #I, issues; and #SD, static dependencies between entities. 28
3.2 Target System's Architectural Metrics Growth (%) after revealing hidden dependencies. 'D' means dependency 34
3.3 Correlation between dependency count (D(S) and D(S,E)) and the growth of the architectural metrics 34
3.4 Clustered Costs (CC) growth (%) after restructuring using coarse grained entities. #P means Number of Packages, and #C, Number of Clusters . . 35
3.5 Clustered Costs (CC) growth (%) after restructuring using fine grained entities. #CS means Number of Classes, and #CT, Number of Clusters. . 36

4.1 Basic data about target systems. 51
4.2 Target System's Conceptual Cohesion of Co-Change Clusters. 'S' means minimum support; 'C', minimum confidence; 'N': maximum entities per issue, 'CGC': coarse-grained clusters conceptual cohesion, and 'FGC': fine-grained clusters conceptual cohesion. (bold numbers show the selected thresholds, '–' means that the combination did not run in Bunch, '×' means that the ratio of entities in clusters is bellow 1%) 52
4.3 Proportion of entities preserved by the clustering process. #C=Number of Classes, #M=Number of Members, #OC=Original Number of Classes, #OM=Original Number of Members 56
4.4 Top 30 Terms Extracted Using First Strategy (With Frequency Numbers) . 60
4.5 Top 30 Terms Extracted Using Second Strategy (With Similarity Numbers) 60
4.6 Sample Entities Associated With the Cluster 61

Chapter 1

Introduction

Software maintenance is well known as one of the most relevant and challenging tasks of software engineering, in particular for legacy systems. In this situation, developers must spend a significant effort towards software comprehension, which also consists of relating the concepts and features to the modular decomposition of a system. This activity becomes hard for several reasons, such as: (a) the features of a legacy systems are not always available— due to a poor documentation, for instance; and (b) the notion of *modular decomposition* in software engineering presents different meanings. According to a David Parnas' seminal paper [65, 28], developers and practitioners should think about modularity as task assignments, though many works relate modularity to language constructs, such as C++ namespaces, Java packages, classes in object-oriented languages, or aspects in aspect-oriented extensions. This misunderstanding often leads to concerns that are either crosscutting through different "modular" unities or concerns that are tangling among other concerns, when considering the same "modular" unities.

Several attempts to reproduce the essence of modular unities as tasks assignments have been presented for software engineering. For instance, Kersten and Murphy introduced the *degree-of-interest* model [48], which captures the task context of program elements by monitoring the programmer's activity using Mylar— a set of Eclipse plugins. Although this is an interesting approach for relating modules to task assignments, **it does not solve the reverse engineering problem of mining the modules of a system from an existing code base**, whose development started without the use of Mylar (or another equivalent tool set). Two properties of a software make this reverse engineering problem a particular challenging task: the size of the code base (hundred thousands lines of code) and the lack of adherence to a stable architecture.

This is the case of SIOP[1], the Brazilian Government Budget System developed using the Java Enterprise Edition platform. Recently, the development team started an effort

[1]Sistema Integrado de Planejamento e Orçamento

to guide the evolution of SIOP through a feature driven approach, such as the Feature Oriented Software Development (FOSD) [6]. FOSD is a feature-oriented method for the construction, customization, and synthesis of large-scale software systems. Based on the FOSD definition, a feature is a piece of software functionality that satisfies a requirement, represents a design decision, and provides a potential configuration option [6]. The SIOP's development team chose the FOSD paradigm as part of a software development process revision. The main stated goal of this revision was to improve traceability between code, implementation tasks and requirements. In this way, the development team expects some benefits with this effort for process improvement: provide a communication pattern between stakeholders; facilitate code comprehension; and establish a well-defined model for software decomposition. The notion of features emerged as an adequate instrument to fulfill these goals. Additionally, a number of Brazilian federated states also uses SIOP, but in various different versions, resulting in a family of systems to maintain. Moreover, the current development process, started at 2009, is iterative and incremental and a great part of the development effort is dedicated to add or modify functionality. In this sense, a stepwise and incremental software development (SISD) process is already in place. SISD shares many goals with FOSD [6, 11], which strengthens the decision for FOSD adoption.

However, the SIOP development team does not agree about the set of features of SIOP— since the development team lacks an explicit list of the current available SIOP features. Consequently, this is an obstacle for FOSD adoption. As the size of the application is somewhat large (400 KLOC of Java source code), the manual construction of such a list is not desirable and thus we decided to investigate the following research question:

Is it possible to automatically derive a suitable notion of features from the modular unities of a system?

Accordingly, in this dissertation, we present a novel approach that aims to assist on the semi-automatic identification of features from the modular unities of legacy systems, using techniques based on software clustering. Because of time frame constraints, this work concentrates on investigating the suitability of using software clusters as modular unities that can represent features. The actual feature identification task will be covered in future works.

In fact, the approach taken by this work will be useful for any system, not only for SIOP. Nevertheless, SIOP is one of the target systems in the results presented in following chapters, in conjunction with other systems.

1.1 Purpose of this dissertation

The main goal of our work is to explore the use of software clustering as a technique to assist on software comprehension tasks, specifically to recover concepts which can refer to features. The data sources we used in this exploration is mainly the evolutionary data from version history and issue tracking systems. The specific goals are:

- develop a method that builds on existing algorithms and tools such that a development team can follow in order to produce useful information about concepts from a problem domain, extracted from source-code artifacts;
- conduct an empirical study to asses the relationship of extracted clusters with the existing modularity structure of a software to verify if they are similar; and
- conduct an empirical study to asses the conceptual cohesion of extracted clusters in order to reveal if they are suitable to gather related high level concepts, such as the features of a legacy system.

1.2 Rationale

Software maintenance represents an important aspect of the total cost of software during its entire life. It can be viewed from a software evolution perspective, such that it can be defined as a continued development. In that sense, an existing software never stops to evolve, and its complexity tend to grow [21]. For an agile development process, where continuous and iterative development is the norm, this issue has even more significance.

Besides, feature modeling has a major role in FOSD, and for existing software systems, their adoption implies the construction of a feature model. In conjunction with Software Product Line Engineering, and for software systems where the feature variability is a key aspect, they provide methodologies that produces a family of software products at lower costs, in shorter time, and with higher quality [66].

Thus, the development of effective methods and tools that support the recovering of problem domain concepts from existing artifacts has great value. Moreover, tools that allow tracing from implementation pieces to features can provide an important aid in software comprehension as well as the development of software product lines using an extractive approach.

1.3 Outline

The remainder of this work is organized as follows:

- Chapter 2 reviews the essential literature which this work is based on, that is Software Modularity, Feature Oriented Software Development, Mining Software Repositories, Software Clustering, and Mining Source-Code Change History.
- Chapter 3 contains a paper (to be submitted) that explores the modular properties of the software clusters based on evolutionary data. Its goal is to investigate the impact of revealing the evolutionary coupling in selected architectural metrics, and to verify if the clusters can be used as the main decomposition strategy for modularity.
- Chapter 4 contains a paper that has been accepted for publication at the 29th Brazilian Symposium on Software Engineering (SBES 2015). It explores the conceptual cohesion of clusters based on evolutionary data [64]. Also, here we extended the original paper with a new section (Section 4.6), which includes a preliminary discussion about terms extraction from the vocabulary contained in clusters.
- Chapter 5 presents the final remarks and future work.

As Chapters 3 and 4 contain papers, the structure of them reflects the assumption that they are self-contained. For this reason, some concepts and related work presented in Chapter 2 also appear in Chapters 3 and 4. But, these concepts are presented in summarized form in the papers, in the background section of each chapter, while in Chapter 2 they are presented as an extended version. In addition, note that Chapter 2 presents the basic concepts and the papers present more specific information. Thus, we expect an small overlapping between the chapters of this dissertation. Besides that, Sections 3.4.1 and 3.4.2, from Chapter 3, and Sections 4.3.1 and 4.3.2 from Chapter 4, are similar (but not identical), because they contain information about common procedures we follow in both papers. Also, the abstract of the papers were converted in sections named "Chapter Abstract".

Chapter 2

Literature Review

This chapter presents the topics which this work builds on, namely: Modularity, Feature Oriented Software Development, Mining Software Repositories, Software Clustering, Mining Source-Code Change History, Reverse Engineering Features using Clustering, and Source-Code Semantics. In the beginning, the modularity concept is introduced, since one of the goals of this work is to verify the relationship between modularity and clusters based on evolutionary data. Following, it is presented the feature oriented software development paradigm, since one of the motivations for this work is to easy the adoption of a FOSD approach in legacy systems. Next, we present the topics related to mining software repositories, and software clustering which we used in the remaining of this work. In the end, we show the techniques which are proposed by some authors to measure the semantics of source-code, which allow us to measure the conceptual cohesion of the clusters.

2.1 Modularity

The concept of modularity is present in a vast number of fields which deals with complex systems [9], including but not restricted to software. Besides that, the concept of modularity is closely related to the concepts of design and artifact. The definitions for these concepts, adopted by this work, are:

- *Artifact*: object produced by the intelligence and human effort.
- *Design*: process of inventing artifacts which have specific functions.
- *Parameter*: units of analysis that form a design structure.
- *Modularity*: a particular pattern of relationships between elements of a set of parameters, tasks and people.

From the concept of modularity, we get the concept of module:

> Module is a unit whose structural elements are strongly interconnected, and, for the other side, weakly connected to elements from other units [9].

Therefore, modularity can be understood as a strategy of design for complex systems, that is, those which can not be created—or fully understood— by a unique individual, efficiently.

This way, from the concept of modularity we can derive another idea, embodied by the terms: *abstraction*, *information hiding*, and *interface*:

> A complex system can be managed by splitting it in smaller chunks, and seeing each one separately. When the complexity of one of their elements cross a certain threshold, this complexity can be isolated by defining a separated abstraction which have a simple interface. The abstraction *hides* the complexity of the element; the interface specifies how the element interacts with the bigger system [9].

From the time that a modular design is intended, the tasks of build of the modules also becomes "modular", once the efforts to build the modules can be distributed to several individuals.

A design consists on taking decisions about parameters which govern the product. The modularity of a design is reached by the maximization of the confinement of these decisions inside the modules. Some parameters are, by their nature, shared by two or more modules. To reduce the potential conflict provoked by the existence of this kind of parameter, we can reason about the decisions that affect them in first place, thus, we create "design rules", which govern the whole process of the remaining design.

During the process to design design rules, it is necessary to decide about the information which will be visible to more than one module, and the information which will be hidden. The information hiding principle was first proposed in software engineering by Parnas [65]. Design parameters which are not visible form the hidden information of the design [9]. Parnas concluded that, if the details of a certain code block are hidden from another blocks, changes in one block can be made in an isolated way.

Beyond the seminal work of Parnas [65], other pioneer authors explored the decomposition in software building. Wirth [76] and Dijkstra [30] consider the activity of programming as a sequence of decisions of design, which leads to the successive refinement of software by the addition of details. This approach leads to a modular design in the sense that the decisions stay hidden inside each refinement.

In software architecture literature, module is an implementation unit, comprising well-defined responsibilities and that can be the basis for task assignments, which should be defined according to the information hiding and to the separation of concerns principles [10, 24]. This definition is aligned with the Parnas proposal.

After the work of Parnas, several language constructions were proposed, to support modularity [28], such as C++ namespaces, Java packages, and classes in object-oriented programming languages. However, we do not have consensus about modularity in the software engineering field yet. Parnas, in a panel from 2003, stated:

> My previous work clearly states modularity as a design problem, not a programming language problem. A module is a task assignment, not a subroutine or another language element. Although some tools can make the job easier, no special tools are necessary to fulfill the main goal, just discipline and ability [28].

Despite these new proposals centered in programming languages, some concerns still resisted to the confinement promoted by these proposals. Thus, other researchers desined new proposals which involve either new language elements or new tools, such as: aspects, monads, mixin layers, and multidimensional separation of concerns [28].

These innovative approaches share at different levels the notion that no particular unique decomposition is capable to express in a full modular way all concerns of a software. In this sense, the implementation elements of a programming language can be associated to different modules, each one with a distinct nature.

In a more recent work, Kersten and Murphy [48] proposed a model to capture the task context assigned to a developer, associating to that context the program elements scattered in a codebase. This model, known as *degree-of-interest* (DOI), has the goal of assisting the developer in the task of to locate the code associated to a task assigned to him/her. *Mylar* is a tool that implements the DOI model and was built as an Eclipse plugin. Mylar monitors the developer activities and shows the DOI model to developers in an Eclipse view. This proposal embodies both the Parnas perspective on modularity as task assignment and the vision that more then a modular decomposition exists for the same code base, since one program element can be part of more than one DOI at same time.

2.1.1 Design Structure Matrix

Related to other disciplines, as Engineering and Project Management, there is a set of tools used for reasoning about a particular design, and to formalize operations which transform a design structure in another. Steward [72] proposed one of the most well known of these tools, *Design Structure Matrix* (DSM)[1], which is widely used to capture the level of dependency between different parts of a system [8].

According to Steward [72], design involves the specification of many *variables*, and a precedence order of the variables. We can think of a variable as an element, decision

[1]DSMs are also known as *Task Structure Matrix* (TSM)

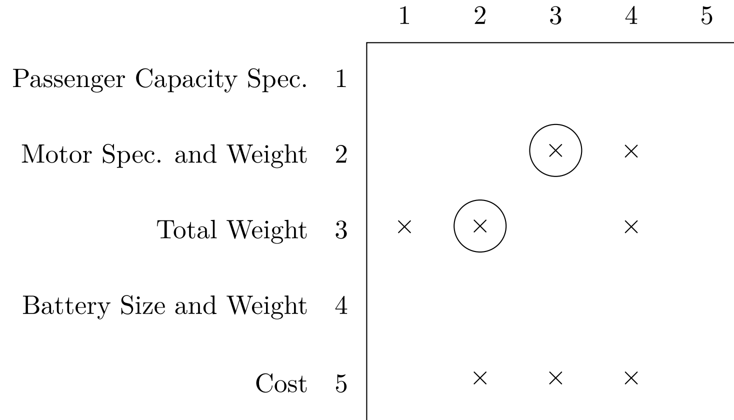


Figure 2.1: Precedence Matrix (adapted from [72].)

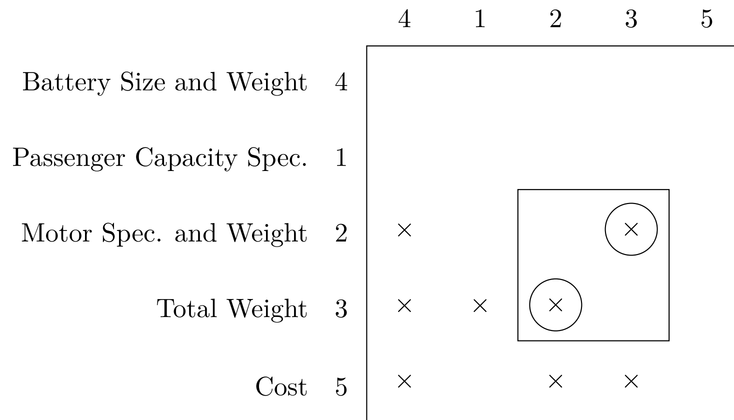


Figure 2.2: Precedence Matrix with Partitions (adapted from [72].)

or task. When a variable A cannot be determined unless B is first known or assumed, and B cannot be determined unless A is first known or assumed, we have a *circuit*. The usual engineering tools, such as Critical Path Planning, does not handle circuits, although DSMs are able to deal with these situations. Thus, Steward proposed a *Design Structure System*, which is used to produce a DSM from the dependency table of the variables.

The first step to construct a Design Structure System is to determine a *precedence matrix*. Figure 2.1 shows an example which represents the design of an electric car. The '$\times$' in a cell i, j, represents a dependency from row i in relation to col j. The circles represent circuits. The next step is to reorder the variables to make the matrix more "triangular", i.e., move both row and column of a variable such that most of the '$\times$' marks are bellow diagonal. Then, the circuits and the dependencies above diagonal must be traced by partitions, as shown in Figure 2.2. The last step is to estimate values for the variables in circuits, and after to reorganize the matrix to leave above diagonal only variables with values estimated. This process leads to a modular DSM.

Eppinger et al [34] then proposed some extensions to DSM representation, including

measures of dependency and task duration, and extensions to the ordering procedure, by breaking the variables into parameters and recombining them into new variables, in order to produce smaller partitions.

DSMs can be also used while designing software [55]. In this case, the elements could represent software artifacts, or some other syntactical element. The dependencies can be any kind of coupling between software elements, both statical or dynamic, such as usage dependencies or inheritance, and the partitions can be packages, namespaces or similar.

2.2 Feature Oriented Software Development

According to the goals of this work, one of our main interests is the software decomposition based on features. In software engineering, the definition of the term *feature* is not a consensus. Table 2.1 shows the definitions compiled by Classen [23].

Table 2.1: Definitions for the term *feature* found in literature

Author	Definition
Kang et al.	A prominent or distinct aspect, quality or characteristic, visible to the user of a software system or systems [44].
Kang et al.	Functional, distinctively identifiable abstractions which can be implemented, tested, delivered and maintained [45].
Eisenecker and Czarnecki	Anything that users or client-programs can want to control with respect to a concept [26].
Bosch et al.	A logic unit of behavior specified by a set of functional and non-functional requirements [20].
Chen et al.	A product characteristic from the viewpoint of the user or the client which, in the essence, is formed by a cohesive set of individual requirements [22].
Batory	An enhancement or increase of an entity which introduces a new service, capacity or relationship [12].
Batory et al.	An increase in the product functionality [13].
Apel et al.	A structure which extends and modifies the structure of a given program with the goal of satisfying the requirement of a stakeholder, to implement and encapsulate a design decision, an to offer a configuration option [7].

From top-down, definitions become less abstract and more technical. While the first five definitions say that features are abstract concepts from the application domain, used to specify e distinguish software systems, the last three definitions capture the fact that features must be implemented to satisfy a requirement [6].

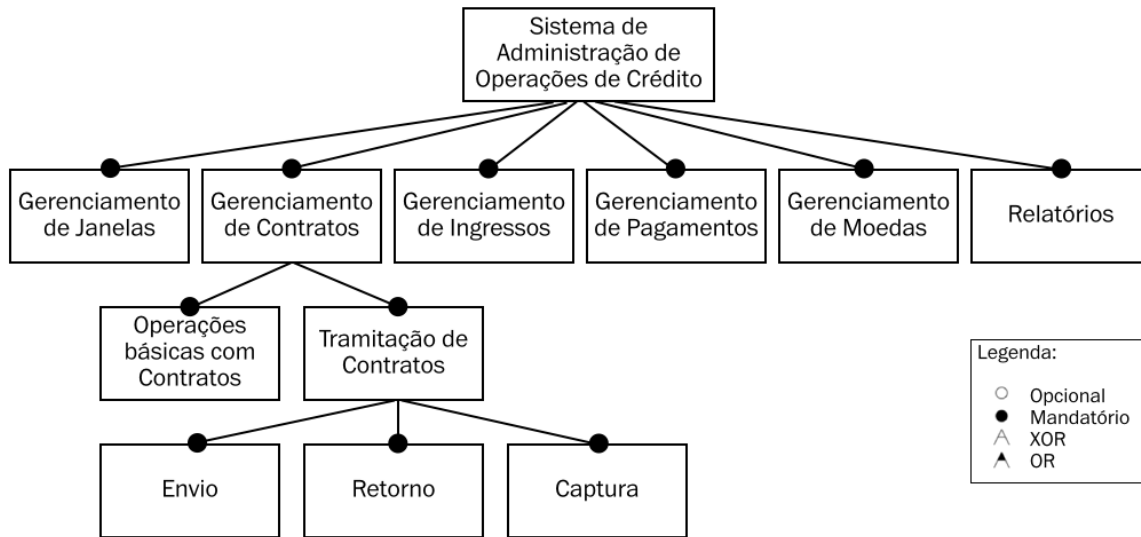


Figure 2.3: Feature model for a subsystem belonging to an important Brazilian Financial System

In this work, we will adopt the definitions of *feature* given by Batory and Apel, since they are closer to the notion of association between features and implementation tasks, and how they approximate to the features role in the software decomposition and its design.

The seminal work of Kang et al. [44] was the first to introduce the feature concept to describe the variabilities and commonalities of a set of software systems. They introduce the *feature model* concept, which describes the relationships and dependencies of a feature set belonging to a particular domain [6]. Figure 2.3 shows an example of a common notation for feature models.

A common scenario for use of features and feature models is the building of *software product lines* (SPL). A SPL is a set of software intensive systems that (a) share a set of features and satisfies the specific needs of a business segment or mission and (b) are developed from a common set of assets [25]. However, in the SPL development, the role of features is not necessarily central. Many SPLs are designed thinking on features, but implements without making the features explicit [6].

The feature concept is central for the feature-oriented software development (FOSD) paradigm, which is used to build, customize, e synthesize large scale software systems. The basic idea of FOSD is to decompose the software system in terms of features. The goal of this decomposition is to built well-structured software that can be adapted to the user needs and to the application scenario [6].

FOSD is a paradigm that provides the systematic use of features in all phases of the software life cycle. Features are used as first-class entities to analyze, design, implement,

customize, debug, and evolve a software system. That is, features not only emerges from a software system structure and behavior, but also are explicit and systematically used to define variabilities and commonalities, to promote the reuse, and to structure the software according to these variabilities and commonalities [6].

FOSD has a substantial overlap in relation to other software development paradigms [6]. The main differences in relation to some popular alternatives are:

- *Stepwise and incremental software development.* The idea of this paradigm is to encapsulate individual development steps which implement distinct design decisions. The goal of this approach is to structure the software such that it would support changes. FOSD shares this goal. In fact, FOSD expanded the development of these initial ideas in the context of large scale software synthesis and of software product line [6].
- *Aspect oriented software development* goal is to modularize crosscutting concerns. It was observed that feature frequently are crosscutting concerns, this way, features implementation can benefit from aspect oriented techniques. FOSD goals are different, but it is believed that in some point both paradigms will be hard to distinguish at implementation level [6].
- *Component based software engineering* goal is to build software systems on demand using already built components. Software oriented architecture is a modern instance of this vision. The main difference from FOSD is that components/services are black boxes. It was observed that features frequently are program slices, i.e., crosscutting concerns. In other words, features does not align well with the decomposition imposed by component models. If components are used anyway, there will be a non trivial mapping between features and components [6].
- *Software product line and domain engineering* are paradigms whose subjects are software systems families instead of unique systems. FOSD is a software development paradigm which can be used to develop software product lines and domain engineering. However, product lines and domain engineering are not limited to FOSD [6].

Feature implementation is hard when represented in codebase, usually because of the lack of explicit elements at programming language level. Some techniques can be employed to cope with this problem, like mixin and collaboration based design, to separate the feature related code from the base program. Other ideas about modularity, which were proposed before the feature implementation research, can be used. They have as a goal in common the software modularity at a larger scale than function or classes [6].

2.3 Mining Software Repositories

Information present in version control systems, bug tracking, communication files, install logs, and code, characterize software repositories which are commonly available to the majority of software development projects. The Mining Software Repositories (MSR) is a research theme that analyzes e crosses the available data from these repositories to reveal information both interesting and useful about software systems [40].

Software engineering researchers conceived and experimented a broad spectrum of approaches to extract pertinent information and to discover relationships and trends from repositories in the context of software evolution. This activity is similar (but not limited) to the field of data mining and knowledge discovery, therefore the use of the term MSR [43].

MSR corresponds, is this work, to the use of techniques of data mining (or similar to data mining) to examine the software changes and evolution, i.e., we suppose the investigation of multiple versions of the same artifact. This contrasts with the approaches that primarily investigate an unique version of a software system, and which use mining techniques only for analysis.

MSR approaches can be categorized in four dimensions [43]:

- Used repositories: version control systems, defect tracking systems, and communication files. Three categories of information can be mined from these repositories: the artifacts and their versions, the differences between the artifacts and their versions, and the metadata about the change in software.

- Purpose: to analyze system growth, to associate source-code entities in relation to their common changes, or components reuse, for example. Generically, there are two classes of questions in MSR: the first is of *market basket* kind, the other relates to getting metrics.

- Methodology: given one or more repositories and the purpose, a method must be adopted or conceived to answer to questions. Basically, two strategies exist. One is the *properties changes*, which compares the properties of the artifacts versions looking for changes identification in global properties of a software system. Another strategy is to study the processes or facts which influenced the software system evolution from one version to another.

- Evaluation: two verification metrics, *precision* and *recall*, from the information recovery community, are widely used to evaluate MSR tools. Another approach is to use information theory to evaluate probabilistic models.

There are several MSR methodologies with their respective purposes. For example, we can use methods for Clone Detection using evolutionary data, which allow to identify

the changes which frequently occurs and also the clone source analysis [50]. Another methodology is the Frequent Pattern Mining, whose purpose can be the detection of evolutionary coupling, which is a subject of interest in this work. [81].

2.4 Software Clustering

One particularly interesting MSR techniques for this work is *Software Clustering*. This technique is an important discipline in reverse engineering and software maintenance. It deals with unsupervised clustering of software artifacts, like functions, classes, or files, in high level structures like packages, components, or subsystems based on the similarity of these artifacts [16].

The common clustering approaches recover this information directly from the static source-code, using structural dependencies based on, for example, references to variables shared between methods, inheritance, aggregation, and method invocation between classes. Some approaches, enhance the clustering through the use of dynamic dependencies recorded during the program execution [16].

One particular application of clustering in software engineering is on remodularization, which is the reorganization of a codebase into new modules, aiming to fix architectural problems and to isolate coupling. Wiggerts [75] was the first to investigate the problem of remodularization in conjunction with software clustering. His work focused on showing an overview of clustering techniques and to answer the following question regarding software clusters: *what are the entities to be clustered? when are two entities to be found similar? what algorithm do we apply?* The answers were based on the theory available at the time, such as the use of *Jaccard* coeficient as a similarity measure or *k-means* as algorithm.

Anquetil and Lethbridge [5] produced a comprehensive set of experiments with clustering algorithms used for remodularization, using three open source systems and one industrial system. Their research questions are similar to Wiggerts [75] questions. For them, software entities can be files, routines, classes, processes, etc., though their experiments considered only files. Several similarity metrics were tested, based on static dependencies, extracted from the source-code, and based on common vocabulary terms, extracted from identifiers and comments. The metrics were grouped in: association, distance, correlation, and probabilistic coefficients. Then, they tested several clustering algorithms using the Bunch tool [60]. Their main conclusions are:

- Similarity based on common vocabulary terms can produce as good results as static dependencies.
- There is no fundamental difference in choosing hierarchical or non hierarchical algorithms.

- The quantity of information is important, even data of dubious utility can prove useful when used as a complement of other data.

One of the purposes for using this technique is the recovery of the software architecture [37]. One architectural view which is subject of this recovery task is the module view [67], which is considered a group of cohesive software units. However, it must be noted that some current approaches of module recovery use other techniques beyond clustering.

Maqbool and Babri [58] carried out an experiment with several hierarchical clustering algorithms to be used in architecture recovering. They used four open source target systems written in C, and used functions as entities. They found an important conclusion: "the quality of results depends not only on the algorithm and similarity measure but also on the peculiarities of the software system to which the algorithm is applied." (specially with regard to the existence of utility functions).

2.5 Mining Source-Code Change History

In recent years, software engineers become aware of the software evolution as a relevant and underutilized data source to enhance the process of software development and maintenance. Some research groups combined static dependencies extracted from source-code with software evolution data to enhance the clustering result. Other approaches are more focused on software evolution and only work with co-change data (considering artifacts that frequently changed together) [16].

The work of Gall et al. [35] was the first to explore the information from version history repositories to detect co-change dependencies between entities and to suggest remodularization based on such a data. They also call co-change dependencies as logical dependencies and hidden dependencies. These dependencies are called *hidden* because they are not evident in the source-code and can reveal dependencies between entities that are not statically dependent. Nevertheless, their technique, named CAESAR, just reveals the hidden dependencies, and is focused in coarse-grained modules instead of fine-grained entities. They also suggest to refactor modules with a strong logical coupling.

Beyer and Noack [19] introduced the use of co-change dependencies in clustering. First, they defined a *co-change graph*, which is an undirected graph where the set of vertices contains all software artifacts and the set of edges represent dependencies that arise when two artifacts had frequently changed together. An edge is enriched with a weight, representing the count of common changes of the two artifacts. They used a clustering layout algorithm that places co-change artifacts closely together, while the others are placed at larger distances. The algorithm is based on the *Edge-repulsion*

LinLog Energy Model. The evaluation of the method used three open-source softwares, and they concluded that the clusters corresponded to the authoritative decompositions. However, the layout clustering, when compared with conventional clustering, does not provide unambiguous partitions for the entities.

Zimmermann et al. [78] coined the expression *evolutionary coupling* which occurs when parts of the system are coupled by common changes. They asserted that when exists an evolutionary coupling between entities that is not expected to be coupled according to the system architecture, this is an anomaly that may suggest a refactor. Their approach is based on syntactical entities, instead of files or modules. After collecting the evolutionary couplings, dependency matrices similar to DSMs were built and analyzed. Their main conclusions are:

- Fine-grained analysis can be used to detect coupling between functions, methods, and attributes.
- Evolutionary coupling complements static coupling.
- Fine-grained relationships allow a higher precision when understanding commonalities and anomalies.

Chapter 3

Unveiling and Reasoning about Hidden Dependencies Induced by Co-Evolution

3.1 Chapter Abstract

Flexibility is one of the expected benefits of a modular design, and thus "it should be possible to make drastic changes to a module without changing others". Accordingly, based on the evolutionary data available on version control systems, it should be possible to analyze the quality of a modular software architecture— and decide whether it is worth to restructure its design. In this chapter we investigate this issue using a novel approach based on a general theory of modularity that uses design structure matrices for reasoning about quality attributes. We carried out a comprehensive study using our approach and found that unveiling and reasoning about the hidden dependencies of a software design lead to a significant impact on three architectural metrics (*average impact of components*, *system stability*, and *intercomponent cyclicality*). We also investigate the issue of restructuring a software based on co-evolution clusters, and we found that a clustered-based decomposition leads to small improvements on system modularity (7.68% on average). This finding contrasts with previous works [70, 79, 16] that suggest that the analysis of co-change clusters might serve as guidance to developers in the challenging task of redesigning a software.

3.2 Introduction

In a seminal paper about the criteria to decompose systems into modules [65], David Parnas relates module as a work assignment unit and states the well known expected

benefits of a modular design, which encourage the parallel design of each module, the support for reasoning about each module independently, and the flexibility to change each model without the need to change others. Though this notion of modularity relates to work assignment, instead of the decomposition of a software in terms of language constructs (such as Java packages, classes, and interfaces), it is expected that the design structure of a system (i.e. its architecture) should resemble the modularization in terms of work assignments.

The issue about how far the design structure resembles the work assignments has been investigated before, particularly through the mining of evolutionary data available in version control systems (VCS). For instance, Murphy et al. report that more than 90% of the changes committed to the Eclipse and Mozilla source-code repositories involved changes to more than one file, which suggests a typical crosscutting pattern that might compromise software modularity [62]. Likewise, Zimmerman et al. recommend the use of fine-grained evolutionary dependencies (in terms of syntactical entities, instead of files or modules) to reason about software modularity [79]. As a more recent work in this field, Silva et al. use software clustering techniques based on the co-evolution of coarse-grained software elements (packages and classes) to analyze software architectures [70]. As a result, they suggest that it might be worth to restructure the package decomposition according to the results of co-evolution analysis.

The aforementioned works use the source-code history to unveil dependencies that could not be inferred from a static dependency, that is a typical usage relation between software elements. Here we introduce the concept of *hidden dependency* as a special kind of dependency motivated by a set of co-changes between two software elements—given that it does not exist a static dependency between them. Therefore, we say that two entities are evolutionarily dependents when they frequently change together, and, according to a criteria that we introduce later in this chapter, this might lead to a hidden dependency between them.

In this chapter we investigate the impact of hidden dependencies on the architecture using a novel approach that builds upon a general theory of modularity (Section 3.3), which allows us to answer two research questions: (a) *To what extent do the hidden dependencies induced by the co-evolution of components impact the architecture?*, and (b) *Is it worth to restructure the architecture of a system based on the co-evolution clusters?* Answering the first research question serves as a predictor about the capacity of the architecture to accommodate software evolution. Differently, answering the second research question might support future efforts to investigate software reconstruction based on co-evolution clusters (as suggested in [70]). Therefore, this chapter presents the following contributions

- A novel approach for reasoning about the hidden dependencies induced by the co-evolution of software assets (Section 3.4). Differently from previous works, our approach is supported by a general theory of modularity, so that we use a well defined set of tools and metrics for reasoning about modularity.

- A comprehensive study about the impact of co-evolution with respect to the design structure of one proprietary and six Java open-source softwares that we use as target systems. We detail this study throughout Sections 3.5–3.8.

Based on the results of our investigation, we could conclude that the architecture of the target systems do not resemble the work assignment devised by the software evolution. In addition, our experience in this research reveal that the co-evolution clusters might not provide a meaningful unit of modularity on his own. We also relate our research to existing works in Section 3.9 and present the final considerations and future works in Section 3.10.

3.3 Background

In this chapter we use the Baldwin and Clark general theory of modularity [9] to investigate the impact of hidden dependencies. Hidden dependency is an evolutionary dependency between two entities given that it does not exists a static dependency between them. Two entities are evolutionarily dependents when they are frequently changed together. Static dependency is a typical usage relation among source-code entities, such as method call and field access.

The Baldwin and Clark theory considers modularity as an important factor that has been uplifting innovation in different domains, and thus, the evolution of the computer industry, for instance, might be explained through the modular design of computers in the nineteen sixties, when IBM launched the System/360 computer family. The original theory makes use of two main components: design structure matrices for reasoning about the architecture of a system and six modular operators that describe how a system might evolve towards a modular design. More recently, several research works investigate properties of software architectures using elements of this theory (in particular DSMs) [73, 53, 54]. In addition, MacCormack et al. [55] extended the Baldwin and Clark theory to introduce specific metrics for reasoning about software modularity. To answer our research questions introduced in Section 3.2, we use both the visual representation of DSMs as well as several architectural metrics that can be computed from DSMs. In the remaining of this section we introduce these elements.

			1	2	3	4
E_1		1	·			
$E_2 - E_3$	E_2	2	×	·	×	
	E_3	3	×	×	·	
E_4		4			×	·

Figure 3.1: Example of DSM

3.3.1 Design Structure Matrix (DSM)

DSM is a technique for representing the modules and their dependencies in a system. This representation, which helps architects on the design, development, and management of complex systems [33], has been considered one of the most promising techniques for reasoning and measuring modularity [55]. A DSM is often represented as an $N \times N$ matrix, where each row and each column represents an element of a system [33].

Figure 3.1 shows an example of a DSM. The labels E_1 to E_4 identify the elements. Graphically, we use an '×' mark in a cell at row i and column j when the element i depends on the element j. To define the modules of a system, it is usual to have a set of partitions of elements accompanying the DSM. An example of partition is also shown in Figure 3.1, as indicated by the thick borders. In Section 3.4 we discuss two hierarchical partition strategies used in this chapter: (a) the structure of the source-code packages and (b) the clusters based on evolutionary dependencies.

We use DSMs to analyze the design of existing software, even though instead of considering the source file as an element [55], we consider fine-grained entities (including classes, methods, and attributes). In addition, we represent two kinds of dependencies: static dependencies, and hidden dependencies induced by the co-evolution of the software entities. To evaluate the software architecture, we use DSMs together with the metrics we present in Sections 3.3.2 and 3.3.3.

3.3.2 Architectural Metrics

To investigate the impact of hidden dependencies, we use three architectural metrics based on the design structure of a system: *Average Impact*, *System Stability*, and *Intercomponent Cyclicality*. This decision was motivated because these metrics use the dependency structure to estimate how robust a software architecture is to accommodate confined changes.

			1	2	3	4
E_1		1	$\cdot$			
$E_2 - E_3$	E_2	2	$\times$	$\cdot$	$\times$	
	E_3	3	$\times$	$\times$	$\cdot$	
E_4		4	$\times$	$\times$	$\times$	$\cdot$

Figure 3.2: Example of a transitive closure

In addition, they have been previously used and we could use an existing tool (Lattix[1]) to automatically collect these metrics from the DSMs of the subject systems. The technical reports of Lattix define these metrics as follows:

- *Average Impact (AI)* of all elements. For each element, the impact is calculated as the total number of elements that could be affected if a change is made to it. For this metric, the lower, the better.

  To know how many elements are affected, a transitive closure of the original matrix which represents the DSM is built. Specifically, a transitive closure of a DSM $D = [d_{ij}]$ is another DSM $D' = [d'_{ij}]$ where,

$$d'_{ij} = \times, \quad \text{if exists a path between } i \text{ and } j \text{ in } D, \tag{3.1}$$

  and, exists a path between i and j if exists a sequence,

$$d_{k_0 k_1}, d_{k_1 k_2}, \cdots, d_{k_{n-1} k_n}, \tag{3.2}$$

  where, each $d_{k_{l-1} k_l} = \times$, and $k_0 = i, k_n = j$.

  Figure 3.2 shows the transitive closure computed from the DSM of Figure 3.1. Note that each cells $4, 1$ and $4, 2$ were filled with '$\times$'.

  Given the transitive closure, the AI metric is defined as:

$$AI = \frac{1}{|E|} \sum_{i,j=1}^{|E|} 1, \text{ If } d_{i,j} = \times \text{ in the transitive closure.} \tag{3.3}$$

  Where $|E|$ is the total number of elements. Thus, the AI of the DSM in Figure 3.2, is $\frac{1}{4}(0 + 2 + 2 + 3) = 1.75$.

[1]http://www.lattix.com

- *System Stability (SS)i:* measures the percentage of elements (on the average) that would not be affected by a change to an element. It is calculated according to the equation,

$$SS = 100 - \left(\frac{AI}{|E|}\right) 100, \tag{3.4}$$

  For this metric, the closer to 100%, the better.

  For Figure 3.2, the SS is $100 - (\frac{1.75}{4})100 = 56.25\%$

- *Intercomponent Cyclicality (IC)* reports the percentage of elements that are in a cyclical dependency with other elements in other partitions. Attempt should be made to bring this number to zero or close to zero.

  We also compute this metric based on the transitive closure. In Figure 3.2, we have two elements involved in a cyclical dependency (2 and 3). But, as they are in the same partition, the IC for this DSM is zero. Otherwise, it would be $\frac{2}{4} = 50\%$.

3.3.3 Clustered Cost

Besides the architectural metrics, we reason about the partitions strategies (static and evolutionary) using the *Clustered Cost* metric [55], which also derives from the Baldwin and Clark theory. The *Clustered Cost* metric assigns a weight to each dependency, considering this weight should be lower if two dependent elements are in the same partition.

The metric requires the identification of elements with a large number of dependents. These elements are called *vertical buses* in [55]. Given the number of elements N, a element j is a *vertical bus* if:

$$\left(DepRatio(j) = \frac{\sum_{i=1}^{N} 1, \text{ if } i \text{ depends on } j}{N} \right) > bus\,threshold \tag{3.5}$$

Then, given the set of elements E and the set of dependencies $D \subseteq E \times E$, the *Clustered Cost* of a DSM is the sum of the *Dependency Cost* of each $(E_i, E_j) \in D$, as defined bellow:

$$DependencyCost(i, j) = \begin{cases} 1, & \text{if } j \text{ is a vertical bus} \\ n^{\lambda}, & \text{if } i, j \text{ in same partition} \\ N^{\lambda}, & \text{otherwise} \end{cases} \tag{3.6}$$

where n is the size of the partition, N is the size of the DSM, and λ is a user-defined parameter. We discuss the value of this parameter and the *bus threshold* in Section 3.6.

In Figure 3.1, for example, we have:

$$\begin{aligned} &DepRatio(1) = \tfrac{2}{4} = 0.5,\ DepRatio(2) = \tfrac{1}{4} = 0.25,\\ &DepRatio(3) = \tfrac{2}{4} = 0.5,\ DepRatio(4) = \tfrac{0}{4} = 0. \end{aligned} \tag{3.7}$$

Thus, if we assume *bus threshold* $= 0.3$ and $\lambda = 2$, we have the following *dependency costs*:

$$\begin{array}{c} \\ 1\\ 2\\ 3\\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0\\ 1 & 0 & 1 & 0\\ 1 & 2^2 = 4 & 0 & 0\\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \tag{3.8}$$

Then, the *Clustered Cost* of this DSM is $1 + 1 + 4 + 1 + 1 = 8$.

3.4 Methodology

This section describes the methodology (see Figure 3.3) we use to investigate our research questions. To measure the impact of hidden dependencies, and to discover the quality of the decomposition based on co-change clusters, we need to calculate the metrics presented in Section 3.3 (step 5). Before, we need to build DSMs which the metrics will be based on (step 4). As we are interested in both static and evolutionary dependencies, it is necessary to extract the static dependencies from `Jar` files and to compute the evolutionary dependencies using data from VCSs for a given system (step 1, 2 and 3).

The release version for the `Jar` files coincides with the final period of the change history extracted from VCS. In other words, we compute the dependencies between elements contained in the code base at one point in time, using two perspectives: static and evolutionary. While the extraction of static dependencies only uses the code as it was in the release time, to compute the evolutionary dependencies is necessary to accumulate data from all the change history.

To enable the reproduction of this study, we populate a relational database with the results of the first three steps. Besides the use of existing tools discussed in the remaining of this section, we implemented several auxiliary scripts. Both scripts and dataset are available on-line [3, 1].

3.4.1 Extracting Fine-Grained Version History

A VCS repository contains the sequence of changesets applied to the software artifacts. We consider that a *commit* is a changeset that contains more than one artifact at once.

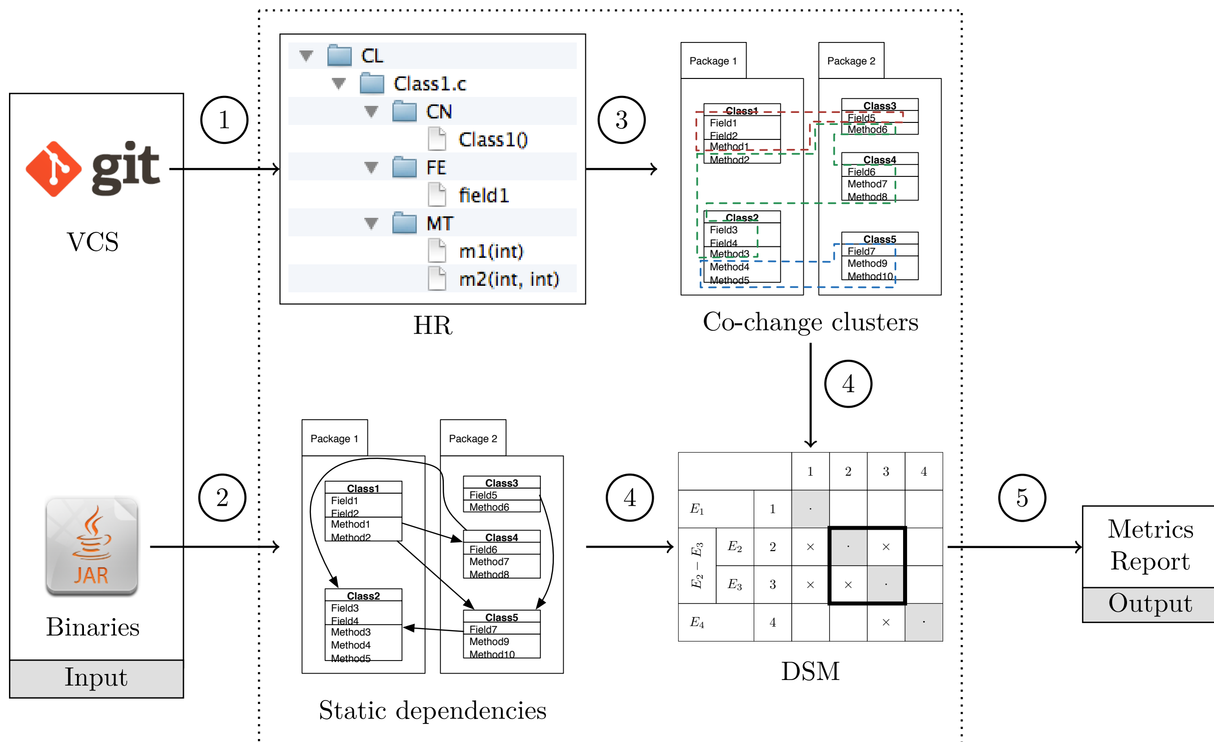


Figure 3.3: Metrics Extraction Process (numbered circles represent the steps.)

Contrasting, a fine-grained VCS repository controls the history of changes applied to some code entities (eg. classes, methods, fields).

Here we are interested in the history of code constructs at the level of classes, attributes, and methods. The goal of this first step is to convert the original VCS repository to a fine-grained repository (see step 1 on Figure 3.3). To this end, we use the *git2historage* (GTH) tool [41] to convert a regular GIT repository into another GIT repository containing the history of the source-code at a fine-grained level, leading to a *Historage Repository* (HR). The input for GTH tool is the original GIT repository and the output is a HR containing the same original commits, and for each commit, its associated artifacts are splitted in the syntactic entities which they contain, i.e., the code of each entity is moved from the original source file to a new file containing only that code (see HR item on Figure 3.3).

3.4.2 Extracting Co-Change Clusters

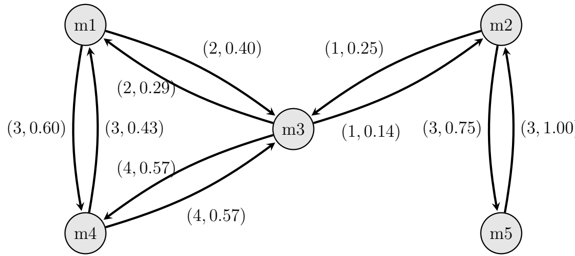
Software clustering technique is a typical approach for discovering groups of code elements based on their mutual dependencies. In general, before applying a clustering technique, it is first necessary to build a *Module Dependency Graph* (MDG), which is a directed graph that contains source-code elements as vertexes and dependencies between them as edges. In our approach, the fine-grained code entities are the vertexes of the MDGs; whereas the

```
public class C1 {
    public void m1() { /* ... */ }
    public void m2() { /* ... */ }
}
public class C2 {
    public void m3() { /* ... */ }
    public void m4() { /* ... */ }
    public void m5() { /* ... */ }
}
```

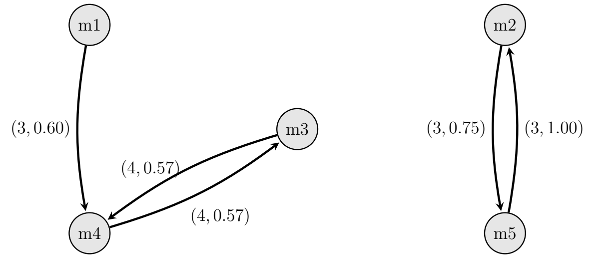
(a) Current source-code

Commit	Description	Entities
028a98d	Issue #1	m1, m3
d8fd425	Issue #2	m1, m3
c90c352	Issue #3	m1, m4
ad3f78a	Issue #4	m1, m4
cd5e305	Issue #5	m1, m4
7de2d7b	Issue #6	m3, m2
83850f6	Issue #7	m4, m3
59561f2	Issue #8	m4, m3
b8e3afd	Issue #9	m4, m3
3bed650	Issue #10	m4, m3
5afa3bb	Issue #11	m5, m2
121192e	Issue #12	m5, m2
44b80e9	Issue #13	m5, m2

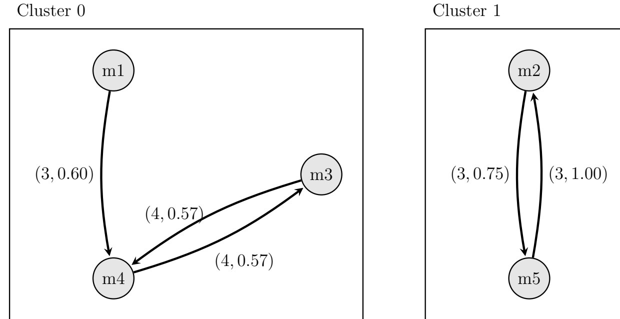
(b) Fine-grained commits from HR



(c) Co-change graph (MDG)



(d) Pruned co-change graph using *minimum support* equals 2 and *minimum confidence* equals 0.5



(e) Co-change clusters

Figure 3.4: Example of co-change cluster extraction (the edges' labels specify *support count* and *confidence* respectively.)

evolutionary dependencies derive the edges of the graph. Figure 3.4 shows an example of co-change clusters extraction, which is also represented by the step 3 on Figure 3.3. The details about it are bellow.

To extract an MDG from the source-code history, we iterate over the commits history and their respective entities stored on HR (Figures 3.4a and 3.4b). In this case, an MDG is a co-change graph $G = (V, E)$, where V is a set of code entities and E is a set of pairs $(V_i, V_j) \in V \times V$, such that exists a commit which contains both V_i and V_j (Figure 3.4c). It is important to note that we followed some guidance about how to

build MDGs from source-code history with the goal to enhance the quality of the clusters. Thus, we do not consider all changesets when building the graph. First, we only consider commits explicitly related at least one issue present in the *Issues Tracking System* (ITS) of the subject software, as we want to resemble the software structure following the work assignments. We assume that the issues stored in the ITS are related to some kind of work assignment: the implementation of a new feature, modification of a existing one, bug fixing, and so on — in fact, the issues' kind does not matter, we use them only to build the MDG. This way, if a set of entities are associated with some issues in common, we conclude that the entities are related with work assignments in common too.

In addition, we follow the suggestion of some authors that a commit should not be considered in the analysis when the number of entities exceeds a threshold [81]. Also regarding our criteria to prune the dependency graph, Zimmerman et al. [79] propose the use of the *support* and *confidence* metrics to measure the quality of the evolutionary dependencies (*support* is called *support count* in [81]). We define *support* as the number of issues in common between two entities.

An MDG must only contain edges with a minimum *support*. To aid the clustering algorithm to produce clusters with better quality, we assign the *support* metric as the weight of each edge, similarly to [70]. The *confidence metric* measures the strength of a dependency between two entities, and is defined as:

$$confidence(V_i, V_j) = \frac{support(V_i, V_j)}{support(V_i, V_i)} \tag{3.9}$$

The term $support(V_i, V_i)$ means the total of issues associated with V_i. Thus, $confidence$ is the proportion of issues where V_i and V_j participates in relation to the total of issues that V_i participates. Again, only edges with a minimum *confidence* will be used (see Figure 3.4d).

Given that criteria above, one must experiment with some thresholds to choose a suitable combination. Some authors gives advice about this, such as Beck and Diehl [16], which state that the values for minimum *support* equals 1 and minimum *confidence* equals 0.4 produced the best results in their experiment. Additionally, they establish 50 as the maximum acceptable number of entities in a commit. On the other hand, Silva et al. [70] use a different set of thresholds, considering the minimum *support* equals 2.

Regarding automation, there is a number of algorithms, methods, and tools to cope with the task of clustering software elements. We choose `Bunch` because of its superior performance on clustering software [60]. `Bunch` applies a heuristic search algorithm based on random elements and increases the reliability of the results by carrying out the clustering process several times.

More specifically, `Bunch` receives an MDG as input and generates a dendrogram (also called *Clustered Graph*; see Figure 3.4e), in which a cluster in a level with less details contains one or more clusters of the next level. In our analysis, we configured `Bunch` with the *Agglomerative Clustering* action and the *Hill Climbing* clustering method as in [16].

3.4.3 Extracting Static Dependencies

In the third step (see Figure 3.3), we extract the static dependencies among source code entities using the *Dependency Finder* (DF) tool [2], as Beck and Diehl [16]. This tool reads a set of binary class files and outputs a report with the static dependencies representing usage relations among entities. We then import those static dependencies into our dataset, using a specific script that associates the entities in the DF report to the fine-grained entities in HR. If one entity from DF report is not found in HR, the dependency is discarded. In general, these entities are code generated by the compiler, such as default constructors and `enum` methods. Thus, there is no prejudice in discard them.

3.4.4 Building DSMs

This is the step 4 in Figure 3.3. To answer the first research question (*To what extent do the hidden dependencies induced by the co-evolution of components impact the architecture?*), we create two DSMs, one with static dependencies and one with both static and hidden dependencies. The elements of both are the fine-grained entities of a system. The partition strategy is based on the original modules (packages) in both DSMs. These two DSMs are then compared, using both visual representation (detailed on Section 3.6) and architectural metrics. This enable us to reason about the impact of revealing the hidden dependencies.

To answer the second question (*Is it worth to restructure the architecture of a system based on the co-evolution clusters?*), we create two DSMs: one DSM that uses the typical *packages* partition strategy; and one DSM that uses co-change clusters as a partition strategy. Each DSM contains both static and hidden dependencies. In this way, we can compare the *Clustered Cost* of the DSM organized by the co-change clusters with the DSM organized by the original modules (Java packages). The difference between these two DSMs is the partition strategy. This setup allows us to investigate which is the superior partitioning strategy (clustered based or based on packages) according to the *Clustered Cost* metric.

3.4.5 Computing Metrics

We collect the architectural metrics using the tool Lattix. It receives as input the DSMs, comprising entities, dependencies and partitions, and outputs a set of metrics values (see step 5 on Figure 3.3).

Note that, the original definition of the *Clustered Cost* metric uses as partitions a set of clusters that are built using an iterative search-based algorithm [55]. Their creators have the intention that the metric is affected only by the amount and the patterns of dependencies in a DSM—and not the quality of a particular partitioning. Differently, in this chapter we measure the quality of a given partitioning based on co-change clusters, and thus we calculate the metric using the partitions based on packages and the co-change clusters instead.

3.5 Study Settings

This section brings details about the study settings we use for investigating our research questions discussed in Section 3.2 using the methodology of Section 3.4.

3.5.1 Target Systems

We selected seven Java projects of different domains as the target systems of our study. All of these project uses either GIT or SVN as version control systems, and an expressive number of commits are related to issues (ranging from 38% for SIOP to 98% for Hadoop). Six of the target systems are open source projects; while SIOP is one fundamental financial system of the Brazilian Government, which the first author of this chapter has contributed to. All of them have large code bases, at least two years of development history, and has been useful for many users.

Table 3.1 summarizes some basic metrics about the target systems. Due to some constraints of `Bunch` regarding the size of the dependency graph [16], we limited the evolution history period of three projects (Hadoop, Eclipse UI and Lucene).

3.5.2 Selection of the Threshold Combination

Several clustered graphs based on different thresholds combinations were computed. For each project, the graphs with best results were used for building the DSMs.

The set of thresholds we experiment are: *maximum number of entities per issue*, *minimum support*, and *minimum confidence*. For the first threshold, we experiment with the value 50, as suggested by Beck and Diehl [16] and the value 100, since we grouped entities

Table 3.1: Basic metrics about target systems. #C means number of classes and interfaces; #F, methods, attributes, and constructors; #I, issues; and #SD, static dependencies between entities.

Name	Description	Version	Period	#C	#F	#SD	#I
SIOP	Brazilian Planning and Budget System	1.26.0	2009/05/14-2014/05/15	4,542	64,700	126,192	2,949
Derby	Relational database	10.11.1.1	2004/08/11-2014/09/10	1,597	22,793	55,447	3,245
Hadoop	Large data sets processor	2.5.2	2013/01/01-2014/09/10	10,351	99,485	70,877	5,331
Eclipse UI	Eclipse main UI	4.5M2	2011/01/01-2014/09/08	7,443	56,649	75,354	9,510
JDT Core	Eclipse Java core tools	4.5M2	2001/06/05-2014/10/21	1,954	30,250	87,356	5,002
Geronimo	Java EE application server	3.0.1	2006/08/24-2013/03/21	3,101	16,476	26,118	1,511
Lucene	Text search engine	4.9.1	2011/01/01-2014/10/27	4,097	28,915	20,236	3,272

per issue instead of commit, and this scenario leads to a greater number of associations. For the second threshold, we experiment with the values 1 and 2, also following the recommendations of [16, 70]. Finally, for the third threshold, we experiment with the value 0 (according to Silva et al [70]) and with the value 0.5 (using a more conservative scenario than the value 0.4 proposed by Beck and Diehl [16]).

The criteria we use to build the MDGs discards some dependencies. As the co-change clusters contains only entities with dependencies between them, the set of entities contained in all clusters is only a subset of the all entities in a system. As consequence, the static dependencies between the entities in this subset are also a subset of all static dependencies of a system. Thus, we can derive the *dependency density* metric, given a evolutionary dependency graph $G = (V, E)$ and a set of static dependencies SD:

$$Density(G) = \frac{|\{(V_i, V_j) : V_i, V_j \in V \wedge (V_i, V_j) \in SD\}|}{|SD|} \tag{3.10}$$

In fact, the set of code entities is subdivided according to the level of granularity — classes and interfaces (thereafter called coarse grained entities), and methods, fields, and constructors (called fine grained entities). There are eight possible threshold combinations that we use to build MDGs. When considering each level, we have sixteen different clustered graphs.

To answer the first research question we only need to use coarse grained entities, because if a fine grained entity F_1 depends on F_2, and they are contained in coarse grained entities C_1 and C_2 respectively, then C_1 depends on C_2. Thus, the effect of revealing hidden dependencies can be captured using only coarse grained entities. For the second question we use the two subsets because we want to understand: (a) the effect of rearranging the classes between different packages, and (b) the effect of rearranging the methods to form new classes.

We selected only two graphs per project to generate the DSMs, one for each level of granularity. The first selection criteria is the higher *dependency density* (with a value of

0.1 at least). In the cases where no significant difference was found (less than 0.05), we use as a criteria (in this order): greater *confidence*, greater *support*, and smaller *number of entities per issue*. We chose *dependency density* as first criteria because, if there were only a few dependencies, the metric values would seem better than they really are. For even cases, the use of higher *confidence* and *support*, and smaller *entities per issue* produces clusters of better quality, as discussed in [16].

After an empirical study, we chose as the best combination the following thresholds: *support*: 1, *confidence*: 0.5, *maximum number of entities per issue*: 50. The only exception was Lucene, that produced better results when the *maximum number of entities per issue* was 100. Several thresholds combinations applied to fine grained entities could not successfully run in `Bunch`, due to the huge number of vertexes of the dependency graph. For this reason, only Geronimo and Lucene produced usable fine grained clustered graphs.

3.6 Results

In this section we present the results of our investigation. We first present an exploratory data analysis that characterizes the target systems according to the scattering of issues and commits throughout the fine grained entities of the subject systems. Then we answer the fundamental research questions of this study.

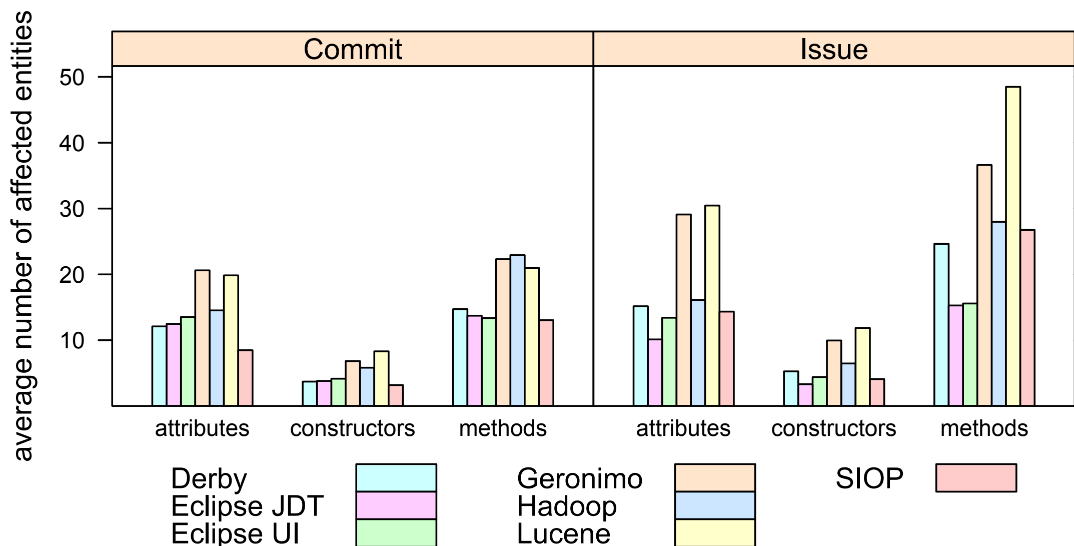
3.6.1 Exploratory analysis of the impact of commits and issues into fine grained entities

We have characterized the subject systems according to the effect of issues and commits into fine grained software entities (attributes, constructors, and methods). Accordingly, we derive a notion of *scattering* that we use in Section 3.6.2 and Section 3.6.3.

Figure 3.5 summarizes this auxiliary result. Considering all systems, each commit affects on average 14.51 attributes, 5.12 constructors, and 17.29 methods. Besides that, Geronimo, Hadoop, and Lucene present an expressive scattering of commits throughout the fine grained entities. In particular, each commit of these three target systems affect on the average more than 20 methods. Regarding the impact of issues into the fine grained entities, on the average each issue of the subject systems affects 18.38 attributes, 6.48 constructors, and 27.90 methods. Once more, Geronimo, Hadoop, and Lucene present a high degree of scattering between issues and the fine grained entities.

There is also a relation between issues and commits, which allowed us to compute the notion of impact discussed above. In one extreme, 30% of the commits are related to issues on SIOP. On the other, more than 90% of the commits are related to issues on

Figure 3.5: Characterization of the systems with respect to the impact of the commits and issues into the fine grained entities.

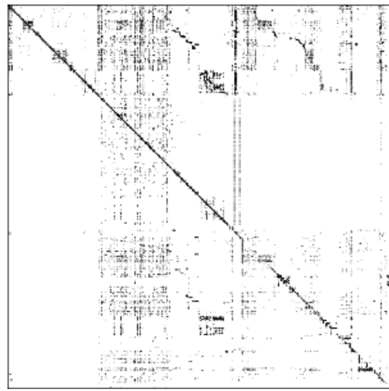


Hadoop. Considering all target systems, each issue requires on the average 1.59 commits. This quantitatively supports a previous assumption that, in open-source projects, there is almost an one-to-one mapping between commits and work assignments [62]. We also consolidated this analysis considering the impact of commits and issues into coarse grained entities, and we found that, on the average, each commit affects 5.10 Java classes or interfaces; and each issue affects 8.81 Java classes or interfaces.

These numbers show that the evolution of the source-code of the target systems, in terms of work assignments, is spread over several code entities. This suggest that it might be worth to reasoning about the architecture of those systems also considering the evolutionary history available on version control systems. Next we further investigate this issue, by empirically assessing the impact of the hidden dependencies motivated by the co-evolution of coarse grained entities into the architecture of the systems.

3.6.2 To what extent do the hidden dependencies induced by the co-evolution of components impact the architecture?

As discussed in Section 3.4, to answer this research question we build two DSMs for each target system. For both DSMs, we use classes as elements and packages as partitioning strategy (as discussed in Section 3.5). Figures 3.6 to 3.12 shows some examples of DSMs for the target systems. The first sub-figure contains only static dependencies; the second contains both static (black) and hidden (red) dependencies.


(a) Static

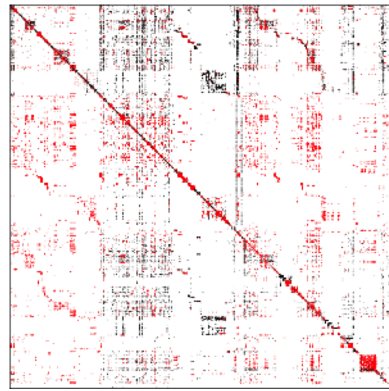
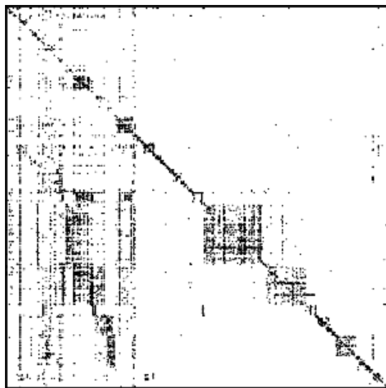

(b) Static and Evolutionary

Figure 3.6: SIOP DSMs


(a) Static

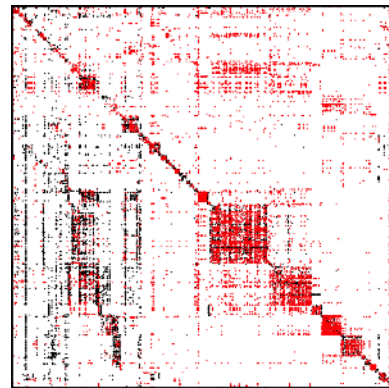
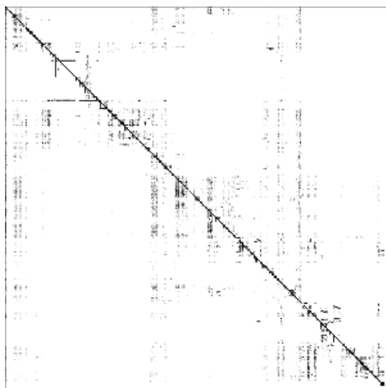

(b) Static and Evolutionary

Figure 3.7: Derby DSMs


(a) Static

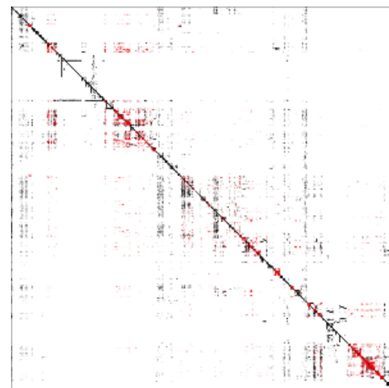
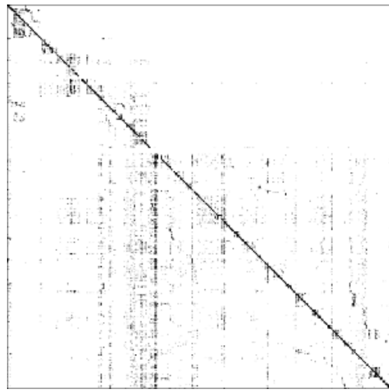

(b) Static and Evolutionary

Figure 3.8: Hadoop DSMs

Considering that in each DSM the rows and columns were sorted by qualified class name (formed by package name first plus class name after), a modular design should


(a) Static

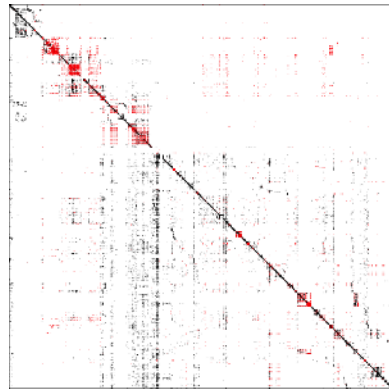
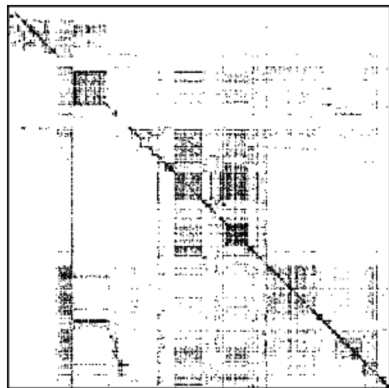

(b) Static and Evolutive

Figure 3.9: Eclipse UI DSMs


(a) Static

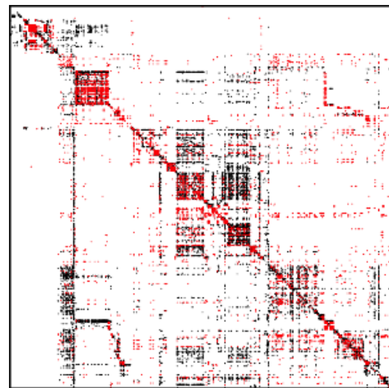
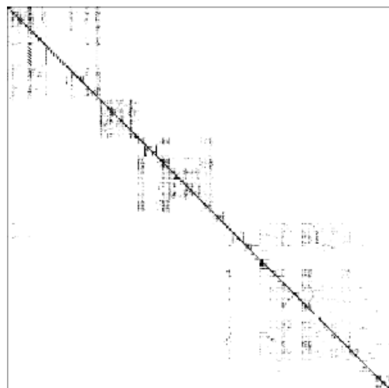

(b) Static and Evolutive

Figure 3.10: JDT DSMs


(a) Static

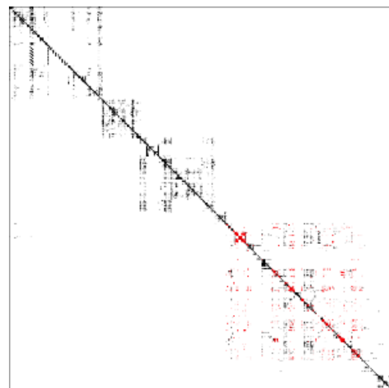
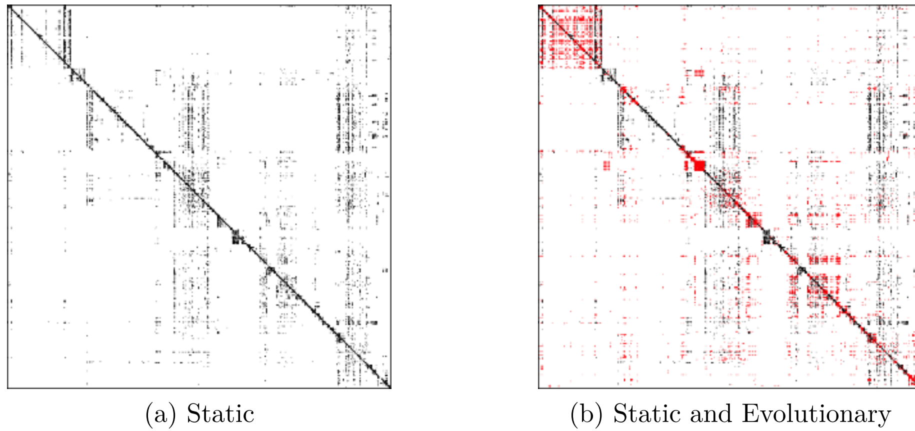

(b) Static and Evolutionary

Figure 3.11: Geronimo DSMs

concentrate most of the dependencies along the main diagonal, as well as along the vertical buses. In this way, comparing the DSMs, different patterns emerge. For instance, Derby,



(a) Static (b) Static and Evolutionary

Figure 3.12: Lucene DSMs

JDT and SIOP present a high degree of scattering of both static and hidden dependencies, which might suggest bad design decisions during the decomposition of these systems into modules. In the case of SIOP, the scattering might also reflects the existing coupling within the organizational structure where SIOP has been developed [55]. Note that, according to our observation that a modular design concentrates most of the dependencies along the main diagonal, we conclude that the other target systems (Hadoop, Geronimo, Eclipse UI, and Lucene) present a better decomposition — which leads to a smaller number of hidden dependencies in these systems.

Indeed, most hidden dependencies in Hadoop, Geronimo, Eclipse UI, and Lucene correspond to the static dependencies, while in the others (SIOP, JDT, and Lucene) there is a prevalence of hidden dependencies. These support the results found by Silva et al. [70], though using a different set of metrics and visualization tool, for the systems investigated in both studies (Geronimo, JDT, and Lucene).

We collected the architectural metrics from the DSMs above mentioned. Table 3.2 presents the growth ratio of the metrics after introducing the hidden dependencies. We consider that the lower the variation, the higher the resilience of a system. In this way, we can reason about the impact of the static dependencies of a system into the hidden dependencies.

For instance, note that (a) most of the Geronimo static dependencies (Fig. 3.11a) are within the main diagonal, and (b) Geronimo is more resilient to the introduction of the hidden dependencies — there is no change on the *system stability* of Geronimo after considering the hidden dependencies (see Table 3.2). Differently, Derby and SIOP present a high degree of scattering of the static dependencies, as well as they are the subject systems with less resilience with respect to the computed metrics. For instance, the introduction of the hidden dependencies increases the *Average Impact* by a factor of

Table 3.2: Target System's Architectural Metrics Growth (%) after revealing hidden dependencies. 'D' means dependency

System	AI Growth	SS Growth	IC Growth	D Growth
SIOP	810	-60	580	150
Derby	540	-70	390	150
Hadoop	540	-20	240	40
Eclipse UI	180	-40	170	40
Eclipse JDT	110	-60	50	70
Geronimo	160	0	80	30
Lucene	760	-50	1300	120

Table 3.3: Correlation between dependency count (D(S) and D(S,E)) and the growth of the architectural metrics

Metric	AI Growth	SS Growth	IC Growth
D(S)	6	63.6	-2.27
D(S, E)	52.7	20.1	1.57

540% and 810% in Derby and SIOP, respectively.

In general, the impact on the architectural metrics after introducing the hidden dependencies is higher on three subject systems: SIOP, Derby, and Lucene. Although Hadoop also presents a considerable growth on the *Average Impact* metric, it is more stable than SIOP, Derby, and Lucene when we consider the other metrics. In addition, there is a small correlation between the number of dependencies (before and after considering the hidden dependencies) and the *Average Impact*, *System Stability*, and *Intercomponent Cyclicality* metrics (see Table 3.3). As a consequence, the impact on these metrics might not be justified by the number of dependencies between software assets, but instead we can conclude that the corresponding lack of resilience is due to the architectural organization of those systems. In these cases, revealing the hidden dependencies caused by the co-evolution of system components brings new perspectives about the software design. For this reason, we consider that the hidden dependencies are relevant from an architectural point of view, and must be considered when deciding about restructuring a software.

Therefore, regarding our first research question, we conclude that the impact of hidden dependencies on the architecture is significant and thus they should be also considered when restructuring a system.

Table 3.4: Clustered Costs (CC) growth (%) after restructuring using coarse grained entities. #P means Number of Packages, and #C, Number of Clusters

System	#P	#C	CC Growth
SIOP	142	88	−7
Derby	120	83	8
Hadoop	273	215	−27
EclipseUI	217	218	−27
JDT	521	407	22
Geronimo	263	109	−29
Lucene	168	75	−16

3.6.3 Is it worth to restructure the architecture of a system based on the co-evolution clusters?

After analyzing the impact of hidden dependencies in the architectural metrics of the target systems, we investigate the effect of a hypothetical re-modularization driven by co-change clusters, using both levels of granularity (as discussed in Section 3.5). First, we built two additional DSMs for each project, both using coarse grained entities as elements. The first DSM uses a cluster-based partitioning strategy, while the second uses a typical package-based partitioning.

In both cases, the DSMs comprise static and evolutionary dependencies. The `Bunch` tool generates hierarchical clusters with different levels of details. We thus decided to select the level of details with greater number of clusters, providing that this number is less or equal the number of leaf packages. Further, we compute the *Clustered Cost* metric. The rationale for using this metric is that it is more sensitive in relation to the actual partitioning, contrasting with the other metrics we used previously in Section 3.6.2 (AI, SS, and IC), that do not take into account the partitions used in a given DSM.

In this chapter, for calculating the *Clustered Cost* metric, we use a $busthreshold = 0.1$ and $\lambda = 2$, as suggested by MacCormack et al. [55]. Accordingly, Table 3.4 presents the resulting *Clustered Cost* measurements. It is possible to note that it is worth to restructure SIOP, Hadoop, Eclipse UI, Geronimo and Lucene according to the clustering partition. For those systems, the average decreasing of the *Clustered Cost* metric equals 17.2% (with a standard deviation of 9.33). For the other systems (Derby and JDT), there is an average increasing of 16.1% on the *clustered cost* metric.

Finally, we built two additional DSMs based on the fine grained entities of two target systems (in a total of four DSMs): Geronimo and Lucene. In the first additional DSM, we use as input the co-change graph involving attributes, constructors, and methods as

Table 3.5: Clustered Costs (CC) growth (%) after restructuring using fine grained entities. #CS means Number of Classes, and #CT, Number of Clusters.

System	#CS	#CT	CC Growth
Geronimo	760	290	−92
Lucene	1,004	1,047	−8

edges. In the second additional DSM, we use the static dependencies graph as input, also considering the fine grained entities of the systems. As an intermediary result, a DSM is built comprising these entities as elements and clusters as partition, and another DSM is built with the same elements as the former, but using classes as partitions. As final output, DSMs' elements were replaced with the most detailed level of partitioning. Thus, for the first DSM, the clusters with the higher level of details become elements, and the immediate superior level was used as partitions. For the second DSM, the classes become elements, and the packages the partitions. Therefore, the finest level of clusters corresponds to an hypothetical set of classes. Table 3.5 shows the metrics we compute from these DSMs.

The data for Geronimo suggests a significant enhancement on the metric, after rearranging fine-grained methods in clusters. However, these values were influenced by the lower number of clusters compared to the classes number. For Lucene, the numbers are more coherent, as the number of clusters is near identical to classes number. The Lucene fine-grained results are better than coarse-grained, but, the gain is concentrated in rearranging hidden dependencies only.

Therefore, regarding our second research question, we can not conclude that restructuring a system according to the co-change clusters is worthwhile, since there is no improvement guarantee on the Clustered Cost *metric.*

3.7 Discussion

This study shows how to measure quantitatively the magnitude of the impact of the hidden dependencies on the architecture. Also, it shows that DSMs can be used to visualize the overall quality of a design and to see the impact of introducing hidden dependencies.

However, the overall result of using co-change clusters alone for reorganization can be negative or have little positive effect. A skilled and motivated team may achieve enhancements of orders of magnitude on *Clustered Cost* metric [55]. However, it seems that these gains can not be achieved by automatically reorganizing the source-code to match the co-change clusters. Fine-grained decompositions seems to have a potential

positive value on design, but the semantic of the clusters must be further investigated in order to see if they are cohesive before any recommendation on that respect.

3.8 Threats to Validity

In this section we present a discussion about some questions that might threat the validity of our work. We organize them according to the internal, construct, and external threats.

Internal Validity. We applied the same method to all target systems, including thresholds. However, we cannot ensure that some combination of thresholds favor or disfavor a particular project. To minimize this effect, we chose the thresholds according to the guidance of previous studies. We also observed a common pattern of influence on the thresholds on all projects, this reveals the independence of the method in regard to them.

As the result of clustering contains only a fraction of the whole set of code entities and dependencies between them, it is possible that certain portions of one project would produce metrics more favorable than of another project. The number of details of the clustered graphs generated by `Bunch` are totally dependent of its algorithm, and thus it can interfere on the metrics values. But this effect is random. The quality of the resulting clusters are also dependent on the performance of `Bunch`'s algorithm.

Also, the quality of our process can be reinforced because the findings were compatible with the work of Silva et al. [70], for the common target systems, even using a different set of metrics and tools.

Construct Validity. While we require an association between issues and commits to resemble the work assignment modules with the clusters, the quality of this association cannot be ensured. In addition, this association limits the number of commits considered in our analysis. Another source of potential confusion is the entangling of commits [42]. These problems can influence the results found.

Also, we have to constrain the history period we analyzed for some projects, due to a `Bunch` limitation. Nevertheless, we mitigate this threat because if the entire period it was used the effect of remodularization would be greater, which would strengthen our finding. The representation of the dependency graph was in conformance with existing works, albeit there are alternatives [18]. Some slight departure from most studies were proposed: the use of entities such methods, fields and constructors as elements; and the issues in common as dependency criteria instead of commits.

External Validity. We selected a small set of seven *Java* projects for this study. This can potentially limit the generalization of our results. We choose a wide range of applications, not limited to open-source ones. All projects had large codebases with a long history of changes. As our purpose is only to give some advice about whether a refactoring

based on co-change clusters is worth, we can expect that the findings be reproducible in some other projects too.

The use of `Bunch` as the only tool for clustering limit the reach of the study. While we can, in the future, add more clustering tools to solve this, the choice for `Bunch` was made after a broad research on the literature, and it was found among the tools that produce the better results for software clustering [57].

3.9 Related Work

This section relates our study to previous research works in four subsections. One additional subsection shows how our work is different from previous works.

3.9.1 Version History and Modularity

The work of Gall et al. [36] was the first to explore the information from version history repositories to detect hidden dependencies between modules and to suggest remodularization based on such a data. Zimmermann et al. [79] proposed an approach to determine evolutionary coupling between fine-grained entities, evolving this approach to predict further changes [81].

3.9.2 DSM and Modularity

Beck and Diehl [17] propose a matrix view to compare disparate software decompositions that is very similar to DSMs. Zimmerman et al. [79] also uses a matrix representation similar to the DSM in their work. LaMantia et al. [52] uses DSMs and the modular operators defined by Baldwin and Clark [9] to analyze the evolution of software systems. Xiao et al. [77] introduce *design rules spaces*, an architecture representation that uses DSMs with both static and evolutionary dependencies for defect prediction.

3.9.3 Clustering and Remodularization

Wiggerts [75] introduced the theory behind the use of cluster to guide remodularization. Anquetil and Lethbridge [5] tested various clustering algorithms and reported their performances. Beyer and Noack [18] introduced the use of co-change dependencies in clustering. Maqbool and Babri [58] assessed various algorithms and parameters for hierarchical clustering to be used for architecture recovering. Some approaches uses semantic clustering to assessing modularity [68, 69].

3.9.4 Co-change clusters and Remodularization

Vanya et al. [74] proposed a semi-automatic approach to suggest remodularization. In their approach, given an initial partition, inter-partitions evolutionary dependencies are identified. Silva et al. [70] propose an approach to assess modularity using co-change clusters. Their method use the Chameleon [46] tool to cluster coarse-grained entities that are compared with the actual package decompositions. According to their work, some deviation can suggest restructuring.

3.9.5 Differences from previous works

This chapter is different from the aforementioned works because our approach 1) is applicable for fine-grained elements, such as: classes, interfaces, methods, fields, and constructors, 2) uses DSMs and metrics from the general modularity theory of Baldwin and Clark [9] for reasoning about the impact of hidden dependencies on architecture, and 3) measures the effect of a remodularization that leads the package structure to match with co-change clusters.

Also, our findings reveal that previous suggestions of remodularization [70], are not applicable in general. Our results show that using DSMs to assess the modularity is viable, and competitive in relation to other techniques [79, 70]. Some of our case studies are also used in [70], that considered issues tracking as well.

3.10 Conclusion

In this chapter we presented a new methodology for reasoning about the hidden dependencies induced by the co-evolution of systems' assets obtained from source-code repositories.

Differently from existing works, our methodology relies on a general theory of modularity. This enabled us to analyze the impact of hidden dependencies on the systems architecture using established metrics and tools. We also investigated the benefits of restructuring the architecture of a system using co-change clusters as a guide. To conduct both investigations, we used seven target systems, six open-source projects and one system from the financial domain of the Brazilian Government.

The results revealed that we can have meaningful insights about the architecture of the systems considering the hidden dependencies used in our approach. The results also showed that restructuring a system using co-change clusters produces little or negative improvements. This contrasts with previous works [70, 79, 16] that argues in the opposite direction.

In future works, we aim at investigating whether the combination of static and evolutionary data as input for software clustering, as suggested by Beck and Diehl [16], produces different results, as pointed out in [38]. We also aim at investigating if local reorganizations, using a subset of co-change clusters, produce better results than when we consider all clusters.

Chapter 4

On the Conceptual Cohesion of Co-Change Clusters

4.1 Chapter Abstract

The analysis of co-change clusters as an alternative software decomposition can provide insights on different perspectives of modularity. But the usual approach using coarse-grained entities does not provide relevant information, like the conceptual cohesion of the modular abstractions that emerge from co-change clusters. This work presents a novel approach to analyze the conceptual cohesion of the source-code associated with co-change clusters of fine-grained entities. We obtain from the change history information found in version control systems. We describe the use of our approach to analyze six well established and currently active open-source projects from different domains and one of the most relevant systems of the Brazilian Government for the financial domain. The results show that co-change clusters offer a new perspective on the code based on groups with high conceptual cohesion between its entities (up to 69% more than the original package decomposition), and, thus, are suited to detect concepts pervaded on codebases, opening new possibilities of comprehension of source-code by means of the concepts embodied in the co-change clusters.

4.2 Introduction

Several approaches for software comprehension have been proposed to help developers to understand the decomposition of a system under different perspectives—instead of limiting the analysis to the typical representation based on the structure and usage relations of the software components. Actually, this static representation introduces several limitations. First, the design structure of a system tends to deteriorate as the software evolves

along the years. Second, it is usual to have a lack of correspondence between the structure and the domain concepts of a system, which are often more related to the tasks necessary to design, develop, and maintain a system. This lack of correspondence leads to the scattering of concepts throughout the components of a system, which hinders developers to answer questions related to traceability, such as *where this conceptual feature is located at the source-code?* or *what features are realized by this piece of code?*

To address the need of different perspectives on software decomposition, Kersten and Murphy propose the *Task Context*, which is created by monitoring the tasks developers carry out during their development activities [49]. Therefore, this perspective captures a notion of decomposition that relates source-code entities to a *conceptual* task structure, which leads to an improvement on productivity [62, 49]. Unfortunately, this approach is not suitable to reverse engineering of system abstractions from an existing code base, since it collects information from the current interactions of the developer with the integrated development environment.

Another approach is to build a representation of the software from the code history. This is the approach followed by Zimmerman et al. [81], which deduces dependencies between software components from the common changes of source-code entities. These are *co-change dependencies*, which were further investigated in other research works [35, 80, 39]. In this way, Beyer and Noack [19], propose the use of *co-change dependencies* of coarse-grained software entities (i.e. object-oriented classes or interfaces) as input for building software clusters, and thus they coined the term *co-change clusters* to label the outcomes of their resulting perspective on software decomposition. Later, Silva et al. [71] empirically evaluate the use of coarse-grained co-change clusters as a software representation and found a mismatch between the cluster based decomposition and the modular structure of the target systems (in terms of Java packages) of four open-source systems.

Other works reason about the quality of a system decomposition using a notion of *conceptual cohesion*, which considers the vocabulary of terms present in software entities to estimate their semantic similarity [56, 59, 51]. Entities with high degree of conceptual cohesion might also be used to construct a different perspective of the software decomposition, named *semantic clusters* [51]. Accordingly, Santos et al. [69], investigated the use of *conceptual cohesion* and *semantic clustering* to assess remodularization.

It is important to note that the works mentioned above propose different perspectives of the software at the coarse-grained level, although the concepts of a software are often disperse at the fine-grained level. In this chapter we investigate this issue using a new perspective of the software that is based on fine-grained co-change clusters. Therefore, the contributions of this chapter are:

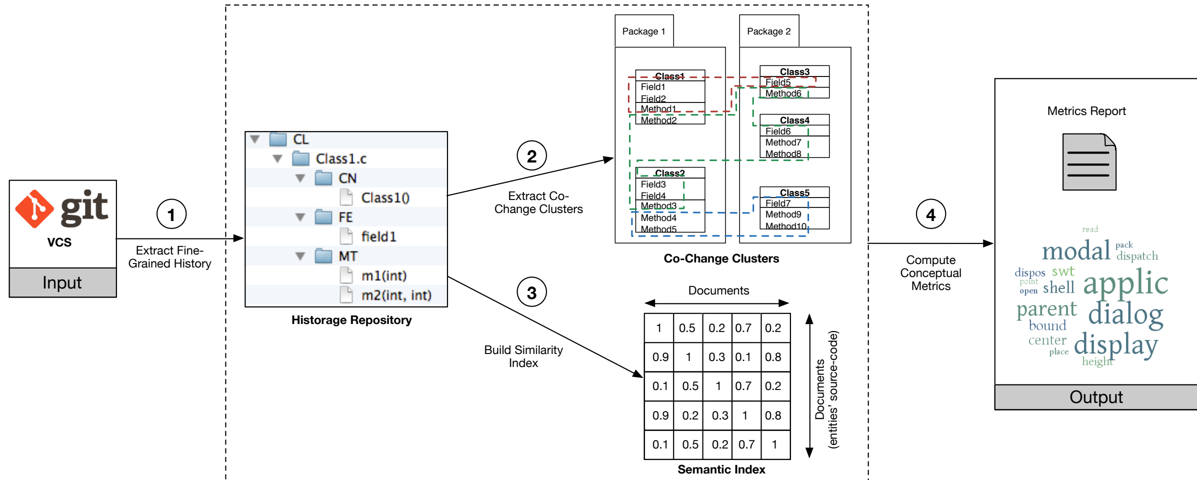


Figure 4.1: Metrics Extraction Process. The numbered circles are the activities, which are executed in order.

- We describe a framework for building a different perspective of a software (based on fine-grained co-change clusters) and a methodology to evaluate the conceptual cohesion of this software perspective (Section 4.3).
- We report the results of an empirical assessment of our software perspective. In this analysis, we compute fine-grained co-change clusters for seven real-world systems and the observations reveal a higher cohesion of the resulting abstractions with respect to the typical decomposition of the systems (Sections 4.4, 4.5, 4.7).

Our findings have several implications. In particular, the terms related to the fine-grained entities that comprise a co-change cluster might serve as input to existing approaches that identify concerns from an initial setting [63]. We discuss some threats to our study in Section 4.8 and relate our research to existing works in Section 4.9. Section 4.10 presents final remarks and future work.

4.3 Methodology

In this chapter we aim at investigating the conceptual cohesion of co-change clusters. First we compute, for each target system, the co-change dependencies and respective co-change clusters using information that is available in Version Control Systems (VCSs) (first and second activities of Figure 4.1). We then compute similarity indexes, beginning from a list of frequent terms used in the source-code entities and computing the similarity between these entities using Latent Semantic Indexing (LSI) [27] (third activity on Figure 4.1). Finally, in the fourth activity of Figure 4.1 we compute a well defined metric for concep-

tual cohesion—as described in [59]. The conceptual cohesion metric is computed for the resulting co-change clusters and the original modular unities (packages and classes).

In the remainder of this section we present more details about each activity mentioned before. To enable the reproduction of our study, the scripts and data set we use are available on-line. [1]

4.3.1 Extracting Fine-Grained Version History

Typically, a VCS repository (such as GIT or SVN) contains the sequence of change sets applied to the software artifacts. In this chapter we consider that a *commit* is a changeset that contains several artifacts, and that a fine-grained repository is a special kind of VCS that controls the history of changes applied to smaller software entities (e.g. classes, interfaces, attributes, methods, and constructors)[2].

Here we are particularly interested in the history of code constructs at the level of these smaller entities, and thus the goal of this first activity is to convert the original VCS repository of a system into a fine-grained repository. To this end, we use the *git2historage* tool [41] to convert a regular GIT repository into another GIT repository containing the history of the source-code at a fine-grained level—a *Historage Repository* (HR). Actually, *git2historage* transforms a conventional GIT repository into an HR containing the same number of the original commits. However, for each GIT commit, *git2historage* splits the related artifacts into a number of fine-grained entities. That is, the code related to each fine-grained entity is moved from the original source file to a new independent file.

The result of this first activity is illustrated as a tree layout of the file system on Figure 4.1, where the source-code of a class `Class1.c` is split on a number of files; one file for each source-code entity. From the HR repository, we build a detailed set of co-change clusters, as follows.

4.3.2 Extracting Co-Change Clusters

In general, the goal of software clustering is to discover groups of code entities considering some kind of mutual dependency and measure of similarity [75]. To apply a software clustering technique, it is first necessary to build a *Module Dependency Graph* (MDG)—a directed graph where: (a) the vertexes are source-code entities, and (b) the edges represent some kind of dependency. In this work we use Java classes and members as source-code entities; and the mutual dependency is based on the entities that had frequently changed together (in fact, the definition of *frequently* is subject to criteria that we will explore in

[1] `http://github.com/mcesarhm/mpca` and `http://goo.gl/3E761S`.

[2] In the remaining of this chapter, for the sake of simplicity, when we refer to classes, we mean classes and interfaces. The same way, when we refer to members, we mean methods, attributes and constructors.

Section 4.4). There are two levels of granularity for co-change clusters: coarse-grained and fine-grained. *Coarse-grained co-change clusters* have classes as vertexes, and *Fine-grained co-change clusters* have members. Here we use a specific kind of MDG: a co-change graph, that is formally defined as $G = (V, E)$, where V is a set of code entities and E is a set of pairs $(V_i, V_j) \in V \times V$, such that there is a least one commit that contains both V_i and V_j. In other words, V_i and V_j frequently change together. To improve the quality of the clusters, we followed several recommendations about how to build graphs from source-code history.

First, we do not use all changesets when building the graph; and thus we only consider commits explicitly related to at least one issue present in the *Issues Tracking System* (ISS) of the target systems—since this decision tends to produce clusters that are more semantically related [71]. With regard to the number of issues associated with commits, we found that only 4% of the commits are associated with more than one issue and only 0.7% are associated with more than 2 issues. For this reason, we might assume a small influence of tangled commits on the results [42, 29].

Second, we assume that a commit should not be considered in the analysis when the number of entities exceeds a given threshold [81]. Commits with too many entities are likely to contain unrelated code, thus we consider them as noise [16]. A code layout change is an example of evolution that might change many source-code files at once. In fact, as the commits must be associated to issues, we adapted this threshold to consider the number of entities per issue. Its specific value is further discussed in Sections 4.4 and 4.5. Finally, we also *prune* the graph using two metrics that measure the strength of co-change dependencies [79]: *support count*, which is the number of issues associated with both entities; and *confidence*, which measures the probability of a change in one entity when another changes. The graph will only contain edges with a minimum *support count* and a minimum *confidence*. The *confidence* metric is defined as

$$Confidence(V_i, V_j) = \frac{SupportCount(V_i, V_j)}{SupportCount(V_i, V_i)} \tag{4.1}$$

where the term $SupportCount(V_i, V_i)$ means the total of issues associated with V_i. Thus, $Confidence(V_i, V_j)$ is the proportion of issues where both V_i and V_j participates in relation to the total of issues that only V_i participates. Note that *Support Count* is symmetric—$SupportCount(V_i, V_j) = SupportCount(V_j, V_i)$, but *Confidence* is asymmetric — $Confidence(V_i, V_j) \neq Confidence(V_j, V_i)$.

So, when considering the criteria above, we simulated several threshold combinations to choose a suitable one, though we also considered some guidance from the literature as discussed further in Section 4.4.2.

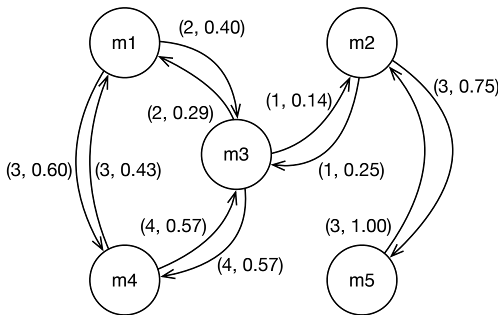
```
public class C1 {
    public void m1() { /* … */ }
    public void m2() { /* … */ }
}
```

```
public class C2 {
    public void m3() { /* … */ }
    public void m4() { /* … */ }
    public void m5() { /* … */ }
}
```

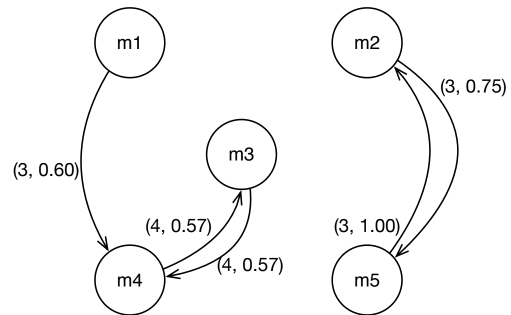
(a) Current source-code

Commit	Description	Entities
028a98d	Issue #1	m1, m3
d8fd425	Issue #2	m1, m3
c90c352	Issue #3	m1, m4
ad3f78a	Issue #4	m1, m4
cd5e305	Issue #5	m1, m4
7de2d7b	Issue #6	m3, m2
83850f6	Issue #7	m4, m3
59561f2	Issue #8	m4, m3
b8e3afd	Issue #9	m4, m3
3bed650	Issue #10	m4, m3
5afa3bb	Issue #11	m5, m2
121192e	Issue #12	m5, m2
44b80e9	Issue #13	m5, m2

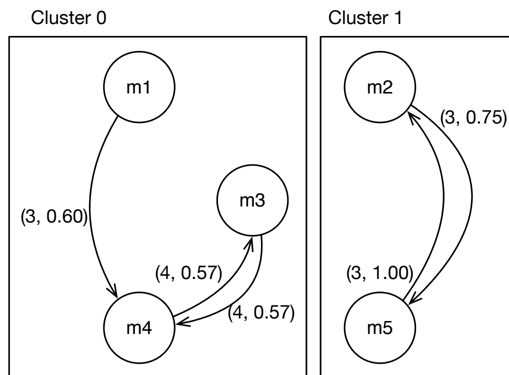
(b) Fine-grained commits



(c) Co-change graph



(d) Pruned co-change graph, using *minimum support* equals 2 and *minimum confidence* equals 0.5



(e) Co-change clusters

Figure 4.2: Example of co-change cluster extraction (the edges' labels specify *support count* and *confidence* respectively.)

Therefore, to obtain a graph from the source-code, we iterate over the change history and their respective entities. Figure 4.2 illustrates this process, considering the source-code of two sample classes. Figure 4.2b also shows the commit log, from the corresponding

HR, for the fine-grained entities contained in these classes. Note that the description of all commits contains a reference for some issue identifier. The column *Entities* are a simplified view of the files present in a specific commit. As we are illustrating a fine-grained cluster extraction, in this case the files affected by a commit represent methods. Figure 4.2c presents a first graph we build from our process, where the vertexes represent the methods and the edges represent the co-change dependencies between them. Each edge has a label in the form (S, C) where S is the *support count* and C is the *confidence*. The edges $m1 \rightarrow m4$ and $m4 \rightarrow m1$, for example, have labels $(3, 0.60)$ and $(3, 0.43)$ respectively. Note in Figure 4.2b that the method $m1$ participates in five commits, and the method $m4$ in seven. In addition, methods $m1$ and $m4$ participates together in three commits. Thus, the confidence values above are 3/5 and 3/7 respectively.

Further, according to the Figure 4.2, we prune that initial graph taking into account the quality criteria that leads to a definition of certain thresholds. In this example, we assumed *minimum support* equals 2 and *minimum confidence* equals 0.5. Figure 4.2d shows the graph with some dependencies removed. The remaining dependencies are those that comply with the *support count* and *confidence* thresholds.

Based on a co-change graph, there are several algorithms, methods, and tools for clustering software entities. In this work we use `Bunch`, an unsupervised cluster tool that applies a heuristic search algorithm based on random elements [60]. `Bunch` increases the reliability of the results by carrying out the clustering process several times. It receives an MDG as input and generates a hierarchical partition set (also called *Clustered Graph*, see Figure 4.2e). In our analysis, we configured `Bunch` with the *Agglomerative Clustering* action and the *Hill Climbing* clustering method. We choose this setup because it produces high quality results in predictable runtime [16]. Figure 4.2e shows the outcome produced by *Bunch* using the graph of Figure 4.2d as input.

4.3.3 Building the Similarity Index

Two steps are related to the activity of building the similarity index. The first step (preprocessing) uses the source-code as input, and outputs a term-document matrix, where terms are words collected from identifiers and comments, and documents are entities' source-code. This matrix is used as input for the second step and then discarded.

In more details, for each entity, we first obtain the last version of the artifacts from VCS and then extract terms from identifiers and comments. During preprocessing, we split identifiers that use camel case naming convention (e.g. for PrintingDevice we get Printing Device), or that is separated by underscores. This is the usual naming convention for identifiers in most popular languages, including *Java*, *C/C++*, and *C#*. We then proceed by removing *stop words*, words with only one character (to remove temporary or index

variables), and words which occurs only once. Next, we reduce the words to their radical (this task is known as *stemming*). Finally, we use the *tf-idf* algorithm to give different weights to the frequent and infrequent terms, in order to compensate their influence [32]. Therefore, very frequent or infrequent words become less important in the computation of similarity index. Then, the original source-code of each entity is transformed into a corresponding bag of words from which we get a matrix with the terms as rows and the documents as columns, and cells representing the presence of terms in documents.

In the second step of this activity (that builds the *Similarity Index*), we use as input the term-document matrix from the first step and produce a document-document matrix, where each cell has the index of similarity between two documents (see Figure 4.1). Here, documents are the entities' bag of words from the preprocessing. In fact, we build two matrices: one for coarse-grained entities, and one for fine-grained entities.

The similarity index is computed using LSI [27], that is an information retrieval technique for measuring the similarity between two documents, a document and a term, or two terms. According to this technique, the documents are modeled in a vector space considering the frequency of its terms. The similarity between two documents is determined by the cosine of the two corresponding vectors. As part of LSI, the dimensions of term-document matrix is reduced into a few orthogonal combinations, using a technique known as Singular Value Decomposition (SVD) [27]. The number of resulting dimensions is informed. When this technique is used for information retrieval, the resulting dimension is between 50–200. Differently, for source-code analysis, existing studies use a number between 20-50 [51].

4.3.4 Computing Conceptual Cohesion Metrics

Similar to other studies that compute the conceptual cohesion between software components [59], we also build the conceptual metrics using the vocabulary present in the source-code entities. The conceptual metrics are a collection of pairs (M, S), where the first element (M) represents a module and the second (S) corresponds to the *average similarity* of the source-code entities contained in the module. We consider two kinds of modules here: static and evolutionary. The static modules are further specialized into classes and packages; and the evolutionary modules into coarse-grained clusters and fine-grained clusters. As modular units, the coarse-grained clusters are equivalent to packages, and the fine-grained clusters are equivalent to classes. Thus, we can generically refer to both packages and coarse-grained clusters as coarse-grained modules; and, for both classes and fine-grained clusters as fine-grained modules.

To compute the similarity of the static modules (both coarse-grained and fine-grained), we retrieve from the HR the last version of the entities' source-code associated with each

module. For each pair of entities of a static module, we retrieve their similarity index from the similarity matrix (that is built in the third activity of Figure 4.1). Next, we compute the module's similarity as the average similarity index of all possible pair of entities belonging to the module, as in [59]. Analogously, we consider the average similarity of the entire system as the average modules' similarity. Likewise, to compute the similarity of the co-change clusters, we iterate over the clusters and their respective entities (obtained in the second activity of Figure 4.1). We compute the cluster's similarity as the average similarity index of all possible pair of entities belonging to the cluster. Thus the average similarity of the entire system is the average of the similarity of its clusters.

Actually, we derive four metrics as the result of this activity. Assuming the similarity indexes of the entities in Figure 4.2 are given by the matrices of Figure 4.3, we can compute the following metrics:

- **Conceptual Cohesion of Packages (CCP)** given the package's classes, and the coarse-grained index, CCP is the average similarity of all pairs of classes. For the entire system, it is the packages' average similarity. Given our sample data (Figure 4.3a), and assuming that $C1$ and $C2$ are declared within the same package, then $CCP = C[C1, C2] = 0.5$

- **Conceptual Cohesion of Classes (CCC)** given the class members, and the fine-grained index, CCC is the average similarity of all pairs of members. For the entire system, it is the classes' average similarity. Given our sample data (Figure 4.3b), $CCC = \frac{1}{2} \times ((F[m1, m2]) + \frac{1}{3} \times (F[m3, m4] + F[m3, m5] + F[m4, m5])) = \frac{1}{2} \times (0.7 + \frac{1}{3} \times (0.8 + 0.5 + 0.7)) = 0.68$

- **Conceptual Cohesion of Coarse-Grained Clusters (CGC)** given the cluster's classes, and the coarse-grained index, CGC is the average similarity of all pairs of classes. For the entire system, it is the clusters' average similarity. Given our sample data (Figure 4.3a), and assuming that $C1$ and $C2$ are within the same cluster, then $CGC = C[C1, C2] = 0.5$

- **Conceptual Cohesion of Fine-Grained Clusters (FGC:)** given the cluster's members, and the fine-grained index, FGC is the average similarity of all pairs of members. For the entire system, it is the clusters' average similarity. Given our sample data (Figure 4.3b), $FGC = \frac{1}{2} \times (\frac{1}{3} \times (F[m1, m3] + F[m1, m4] + F[m3, m4]) + (F[m2, m5])) = \frac{1}{2} \times (\frac{1}{3} \times (0.2 + 0.5 + 0.8) + 0.3) = 0.4$

$$C = \begin{array}{c} \\ C1 \\ C2 \end{array} \begin{array}{c} \begin{array}{cc} C1 & C2 \end{array} \\ \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} \end{array}$$

(a) Coarse-grained index

$$F = \begin{array}{c} \\ m1 \\ m2 \\ m3 \\ m4 \\ m5 \end{array} \begin{array}{c} \begin{array}{ccccc} m1 & m2 & m3 & m4 & m5 \end{array} \\ \begin{pmatrix} 1 & 0.7 & 0.2 & 0.5 & 0.6 \\ 0.7 & 1 & 0.9 & 0.1 & 0.3 \\ 0.2 & 0.9 & 1 & 0.8 & 0.5 \\ 0.5 & 0.1 & 0.8 & 1 & 0.7 \\ 0.6 & 0.3 & 0.5 & 0.7 & 1 \end{pmatrix} \end{array}$$

(b) Fine-grained index

Figure 4.3: Sample similarity indexes computed using LSI.

4.4 Settings

This section brings details about the study settings we use in our investigation, discussing the target systems and the threshold selection we use to improve the quality of the co-change clusters.

This chapter aims to investigate whether fine-grained co-change clusters present conceptual cohesion. It is important to understand if the clusters have been motivated by chance or if they are related to a set of related concepts. To this end, we use some metrics that have been already discussed in the literature [56, 59, 51, 69].

4.4.1 Target Systems

We selected seven Java projects of different domains as the target systems of our study (Derby, Hadoop, Eclipse UI, Eclipse JDT, Geronimo, Lucene, and SIOP). These projects use either GIT or SVN as version control systems, and a significant number of their source-code commits are related to issues (ranging from 38% for SIOP to 98% for Hadoop). Six of the target systems are open source projects; while SIOP is one of the most important financial systems of the Brazilian Government, which the first author of this chapter has contributed to [4]. All of them have large code bases and we could investigate at least two years of the development history of each system (see Table 4.1 for more details). Note that, due to some runtime constraints of `Bunch` regarding the size of the graph used as input [16], we had to limit the evolution history period of three projects (Hadoop, Eclipse UI, and Lucene). Nevertheless, we decided to use Bunch because [16] argues that it presents several advantages over other tools.

4.4.2 Selection of the Threshold Combination

We compute several clustered graphs based on different thresholds combinations. The set of thresholds we experiment are: *maximum number of entities per issue*, *minimum support count*, and *minimum confidence*. For the first threshold, we experiment with the value

Table 4.1: Basic data about target systems.

Name	Description	Version	Period of Analysis	KLOC	Packages	Classes	Commits	Commits Used
SIOP	Brazilian Planning and Budget System	1.26.0	2009/05/14-2014/05/15	521	241	5,611	12,061	100%
Derby	Relational database	10.11.1.1	2004/08/11-2014/09/10	679	215	3,252	6,656	100%
Hadoop	Large data sets processor	2.5.2	2013/01/01-2014/09/10	835	699	11,399	6,864	52%
Eclipse UI	Eclipse main UI	4.5M2	2011/01/01-2014/09/08	617	716	8,370	21,263	13%
JDT Core	Eclipse Java core tools	4.5M2	2001/06/05-2014/10/21	357	78	1,826	16,846	100%
Geronimo	Java EE application server	3.0.1	2006/08/24-2013/03/21	258	744	3,952	4,142	100%
Lucene	Text search engine	4.9.1	2011/01/01-2014/10/27	704	482	4,706	8,854	86%

50, as suggested by Beck and Diehl [16] and the value 100—because we grouped entities per issue instead of commit, and this scenario leads to a greater number of associations. For the second threshold, we experiment with the values 1 and 2, also following the recommendations of [16, 71]. Finally, for the third threshold, we experiment with the value 0 (according to Silva et al. [71]) and with the value 0.5 (a slightly more conservative scenario than the value 0.4 recommended in [16]).

The criteria we use to build the graphs discards some dependencies. As the co-change clusters contain only entities with dependencies between them, the set of entities contained in all clusters is a subset of all entities in a system. Therefore, there is a trade-off involving the use of more conservative thresholds in this case: although it might increase the quality of the clusters, it might also reduce the number of entities considered in the final analysis [16]. As a consequence, we discard a threshold combination when the ratio between the number of entities contained in clusters and the total number of entities of a system is below 1%. As we will discuss in Section 4.5, this ratio tend to be lower for certain target systems, and thus the threshold can not be too restrictive. Nevertheless, it is also necessary to discard graphs with a very low value for this ratio, to prevent from distorting the results.

The set of code entities were subdivided according to the level of granularity—coarse-grained entities, and fine-grained entities. There are eight possible threshold combinations that we use to build graphs. When considering each level, we have sixteen different clustered graphs. Some thresholds combinations applied to fine grained entities could not successfully run in `Bunch`, due to the huge number of vertexes of the graph. This is a known limitation of the tool [16]. Table 4.2 shows the clusters' cohesion per threshold combination. We also compute the similarity indexes in two situations: considering the source-code comments and not considering the source-code comments.

Given the cohesion of each combination the best is the one with greater value. For the results considering comments, *support count* with value 2 produced the best results regarding conceptual cohesion—85% of the combinations with this value were the best. Also, a *confidence* of 0.5 provided 79% of the best combinations. Finally, the *maximum entities per cluster* equals 100, is in 71% of the best combinations. Thus, we assume these

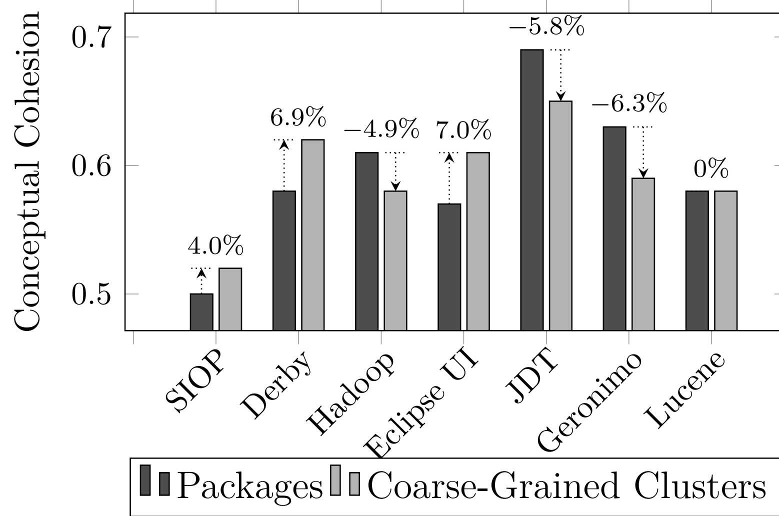
Table 4.2: Target System's Conceptual Cohesion of Co-Change Clusters. 'S' means minimum support; 'C', minimum confidence; 'N': maximum entities per issue, 'CGC': coarse-grained clusters conceptual cohesion, and 'FGC': fine-grained clusters conceptual cohesion. (bold numbers show the selected thresholds, '–' means that the combination did not run in Bunch, '×' means that the ratio of entities in clusters is bellow 1%)

				SIOP		Derby		Hadoop		Eclipse UI		JDT		Geronimo		Lucene	
	S	C	N	CGC	FGC	CGC	FGC	CGC	FGC	CGC	FGC	CGC	FGC	CGC	FGC	CGC	FGC
	1	0	50	0.41	0.39	0.37	–	0.39	–	0.54	–	0.24	–	0.44	0.36	0.47	0.31
	1	0	100	0.38	–	0.45	–	0.45	–	0.52	–	0.33	–	0.24	0.31	0.37	0.35
	1	0.5	50	0.40	0.14	0.54	–	0.47	–	0.55	**0.43**	0.45	–	0.44	0.16	0.48	0.42
with comments	1	0.5	100	0.34	–	0.53	–	**0.58**	–	0.60	0.23	0.19	–	0.49	0.23	0.43	0.39
	2	0	50	0.19	0.31	0.44	**0.58**	0.47	0.24	0.42	×	0.56	0.69	0.55	×	0.52	0.43
	2	0	100	0.37	**0.42**	0.45	0.38	0.51	0.53	0.51	×	0.61	0.46	0.58	0.26	**0.58**	0.60
	2	0.5	50	0.27	0.35	0.31	0.48	0.52	0.61	**0.61**	×	0.62	0.51	0.58	×	0.56	**0.63**
	2	0.5	100	**0.52**	0.27	**0.62**	0.22	0.32	**0.62**	0.60	×	**0.65**	**0.71**	**0.59**	**0.49**	0.41	0.57
	1	0	50	0.41	0.39	0.32	–	0.39	–	0.52	–	0.23	–	0.46	0.36	0.46	0.31
	1	0	100	0.38	–	0.42	–	0.44	–	0.49	–	0.31	–	0.22	0.31	0.36	0.35
	1	0.5	50	0.40	0.14	0.51	–	0.47	–	0.53	**0.43**	0.44	–	0.38	0.16	0.48	0.42
without comments	1	0.5	100	0.34	–	0.51	–	**0.58**	–	**0.57**	0.24	0.17	–	0.52	0.23	0.42	0.39
	2	0	50	0.19	0.30	0.39	**0.58**	0.46	0.24	0.40	×	0.54	0.69	0.57	×	0.52	0.43
	2	0	100	0.37	**0.42**	0.41	0.37	0.50	0.53	0.49	×	0.59	0.46	0.55	0.26	**0.59**	0.60
	2	0.5	50	0.27	0.35	0.24	0.48	0.52	0.61	0.56	×	0.61	0.51	**0.59**	×	0.56	**0.63**
	2	0.5	100	**0.52**	0.27	**0.57**	0.22	0.30	**0.62**	0.57	×	**0.64**	**0.71**	0.57	**0.48**	0.40	0.57

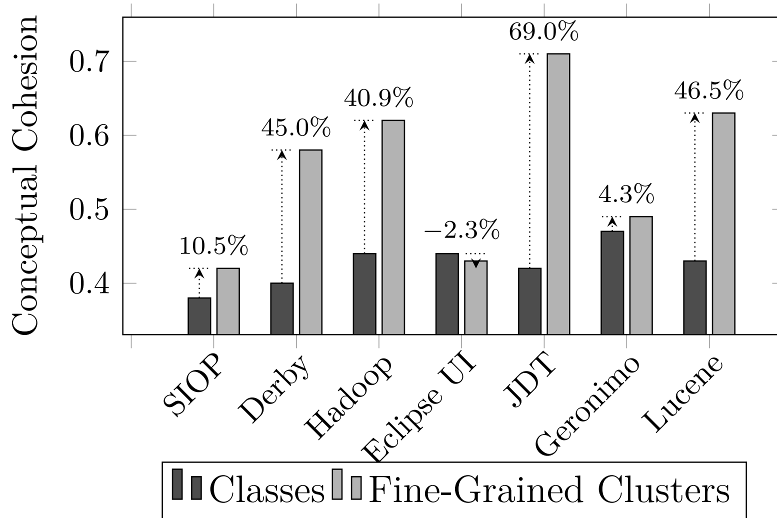
values as the thresholds for *minimum support count*, *minimum confidence*, and *maximum entities per cluster*, respectively. It is important to emphasize that we use *maximum entities per cluster* equals 100, because this is less restrictive than the other considered option (50) that was first suggested in [15]. This can indicate that 50 is too restrictive, causing a loss of conceptual cohesion. The results without comments are near identical of the results with comments. Only 14% of the best combinations have different thresholds when ignoring comments. For this reason, in the remaining of this chapter we will be only reporting results including comments.

4.5 Results

In this section we present the results of our research. For each target system we computed the average conceptual cohesion separately for packages, classes, and clusters. Figure 4.4 shows the average conceptual cohesion for each system (the metrics were introduced in Section 4.3). Figure 4.4a shows data for the coarse-grained modules and Figure 4.4b shows the data for fine-grained modules. The labels near the top of the bars shows the growth (or decreasing) of the conceptual cohesion from the packages or classes to the clusters. Observing this numbers, we can see that, in general, the cohesion of clusters is greater than the cohesion of packages or classes. For coarse-grained clusters, only Hadoop,



(a) Coarse-grained modules cohesion



(b) Fine-grained modules cohesion

Figure 4.4: Target System's Conceptual Cohesion.

Eclipse JDT and Geronimo were significantly outperformed by packages, all of the rest of the clusters have a better conceptual cohesion than packages or classes. For fine-grained clusters, four systems have significant growth of conceptual cohesion, and only one had decreased (Eclipse UI). The other two systems (SIOP and Geronimo) had a low growth. As a future work, we will investigate if this divergence in growth is due to the original decomposition of each system.

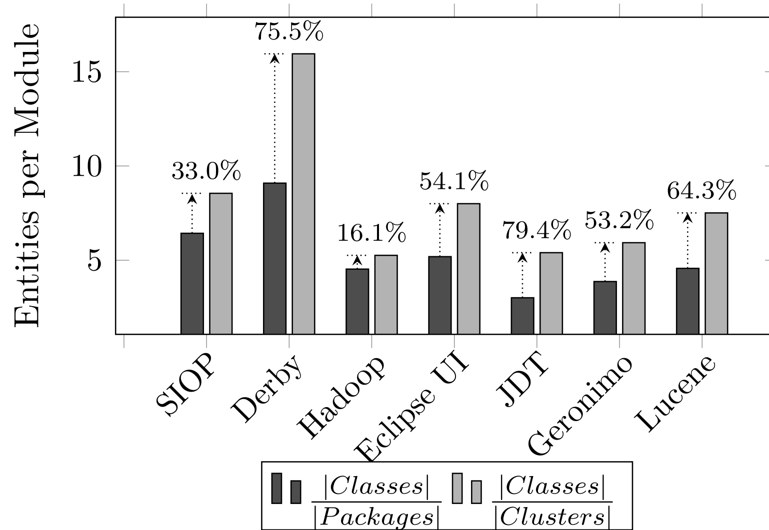
Using the data from Figure 4.4, we can compute the average growth. For coarse-grained clusters, the average is 0.1 (standard deviation 5.9), and for fine-grained clusters, the average is 30.6 (standard deviation 26.5). Therefore, the co-change clusters, in the average, either maintains the same level of semantic when compared with the original organization of the systems (this is the case of coarse-grained clusters), or enhances the

level of semantic (the case of fine-grained clusters). This also suggests that co-change clusters present conceptual meaning, and thus they are not a group of random entities that had changed together. This conclusion is particularly relevant for fine-grained clusters. Although the coarse-grained clusters represent a simple rearrangement of classes into new "packages", the fine-grained clusters represent a real different perspective of the software decomposition, as they break apart classes and form new abstractions with fine-grained entities—that are conceptually related. We believe that this new perspective might help in software maintenance and evolution, as we discuss in Section 4.7.

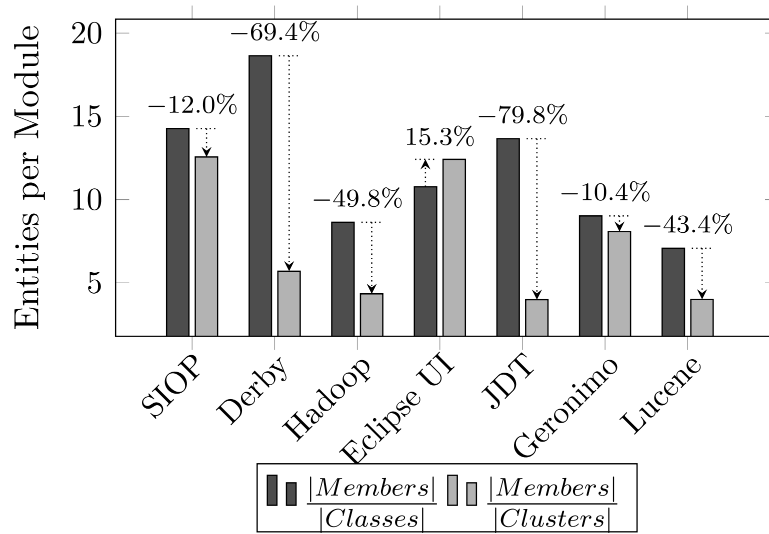
Figure 4.5 shows the average number of entities (classes, members) in each corresponding module (packages, classes and clusters), for each system. Figure 4.5a shows the data for coarse-grained modules, and Figure 4.5b shows the data for fine-grained modules. Here, we consider that the module of a class is either a package or a coarse-grained cluster, and the module of a member is either a class or fine-grained cluster. From that data, we can compute the average growth of entities per module. For coarse-grained modules, the average is 53.7 (std.dev. is 22.7), and for fine-grained modules, the average is -34.3 (std.dev. is 32.7). Comparing these numbers, we can see two different trends: while the number of classes per package tend to be smaller than the number of classes per coarse-grained clusters, the number of members in classes tend to be greater than the number of members in clusters. With regard to the average clusters' size, we have 7.3 (std.dev. is 3.82) for fine-grained and 8.09 (std.dev. is 3.7) for coarse-grained.

We also calculate the correlation (Pearson coefficient) of the growth of conceptual cohesion with the growth of entities per module. In the coarse-grained case, we realize a small correlation between the conceptual cohesion growth and the number of entities per module growth (that is, a correlation value of 0.135). Differently, in the fine-grained case, we realize a *strong and inverse correlation* between the conceptual cohesion growth and the number of entities per module growth (a correlation value of -0.945). Thus, we conjecture that the original classes deal with non-cohesive responsibilities, and that the fine-grained clusters captured their different concepts and form new "classes" that are more conceptually related than the original ones. In that sense, a reverse engineering approach using fine-grained clusters might also lead to a new perspective which locate more easily concepts throughout different classes of a system (and therefore reduces the scattering of concepts), and, thus, can be used to better understanding a software.

It is important to note that we are not suggesting a general restructuring of the software decomposition in terms of co-change clusters. Instead, these clusters provide an orthogonal perspective of the software organization that presents significant improvement of conceptual cohesion (in the fine-grained case). In addition, it is important to note that the computation of co-change clusters discards many dependencies due to the pruning



(a) Coarse-grained modules



(b) Fine-grained modules

Figure 4.5: Proportion of entities in relation to modules.

Table 4.3: Proportion of entities preserved by the clustering process. #C=Number of Classes, #M=Number of Members, #OC=Original Number of Classes, #OM=Original Number of Members

System	#C/#OC (%)	#M/#OM (%)
SIOP	31.10	5.25
Derby	45.79	4.04
Hadoop	29.71	2.71
Eclipse UI	5.83	6.16
Eclipse JDT	20.42	5.34
Geronimo	15.16	1.95
Lucene	32.58	4.03
Average	25.80	4.21
Std.Dev.	13.10	1.51

criteria we use. Accordingly, many code entities are also discarded—since the resulting clusters only include entities with at least one co-change dependency to another entity.

We also investigate the effect of pruning in the proportion of entities preserved in the clusters in relation to the original number of entities in a system (Table 4.3 shows the results of this investigation). The column #C/#OC relates to the coarse-grained clusters, and the column #M/#OM relates to the fine-grained clusters. We can see in Table 4.3 that, on average, the preservation is lower in the case of fine-grained clusters. However, if we consider only the number of entities changed in the period used for computing graphs, these numbers for Eclipse UI raise to 11.02% and 23.79%, (for coarse-grained and fine-grained, respectively); and for Hadoop they raise to 53.1% and 6.06%. The smaller improvement on Hadoop might be due to the fact that the period length used for building the graph is closer to the period length of the whole history, compared with the respective periods of Eclipse UI. Thus, for these target systems, the entity preservation is greater for entities that had recently changed. From a software evolution perspective, these entities are the focus of our interest, because they are more susceptible to change in a near future.

The numbers we discussed above are significant, since in the existing literature about software co-change clusters, a few works brings references about the smaller number of entities preserved after building co-change graphs (for further discussion in that subject, see [16]). Figure 4.6 shows that, on average, entities contained in clusters are more

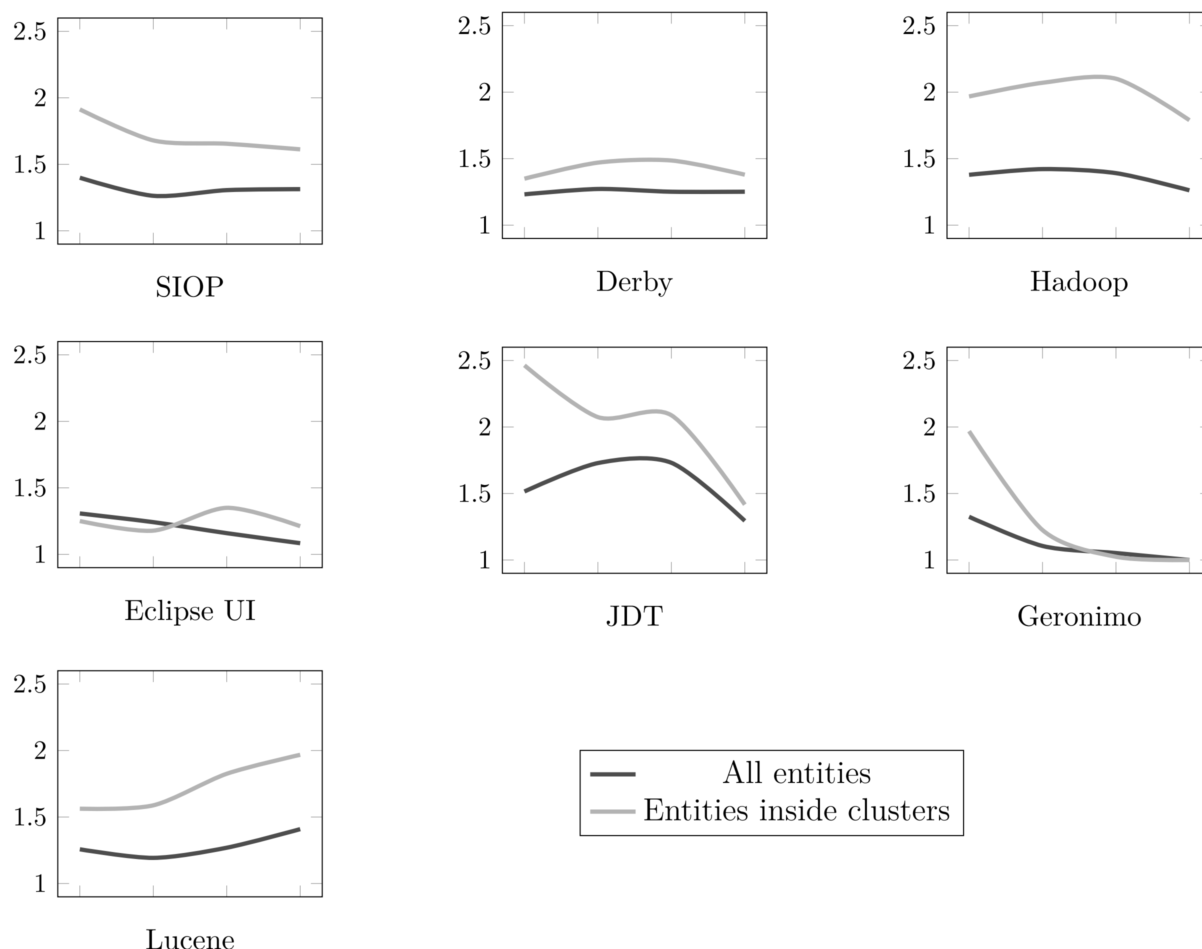


Figure 4.6: Average of commits per entity. X axes represent the last two years of change history for each system, from past (left) to present (right).

frequently changed than the whole set of entities in a system. More specifically, the entities that pertain to at least one co-change cluster, were changed 25% more times than the average of changes for all entities. This data reveals quantitatively that the entities within co-change clusters are more relevant from the evolutionary perspective.

Nevertheless, in the case of this research, this small proportion of fine-grained entities in clusters can be viewed under another perspective: the total number of lines of code they represent. Hence, we computed this measure of size for the entire set of fine-grained clusters for the target systems. We found that, on the average, those clusters represent 31.4 KLOC with a standard deviation of 20 KLOC. Therefore, the size of the fine-grained clusters that present high cohesion could be compared to medium size systems.

4.6 Terms Extraction

Our research revealed that fine-grained co-change clusters have superior conceptual cohesion when compared to both coarse-grained co-change clusters and modular units. Thus, the next question that concern us is: *the terms associated with fine-grained co-change clusters help to identify concepts from the problem domain?*

The idea is to collect the terms contained in the entities bodies and to rank them according to some criterion. As we have seen before, each entity body is converted into a document containing words extracted from identifiers and comments. Thus, we can use the set of documents in the terms extraction task.

Algorithm 1 Extracting terms from clusters

1: $allterms \leftarrow \emptyset$
2: **for** each *cluster* **do**
3: $\quad t \leftarrow \emptyset$
4: $\quad$ **for** each *entity* associated with *cluster* **do**
5: $\quad\quad document \leftarrow$ entity's document
6: $\quad\quad t \leftarrow t \cup terms(document)$
7: $\quad$ **end for**
8: $\quad allterms \leftarrow allterms \cup \{(cluster, sorted(t))\}$
9: **end for**
10: **return** *allterms*

Algorithm 1 shows the basic procedure we took to extract the terms from clusters. Also, we explored two strategies for terms collection and two strategies for sorting metrics. Specifically, we defined two different implementations for function *terms* (line 6) and two different implementations for function *sorted* (line 8).

4.6.1 First Strategy: Terms Frequency

Our first attempt is to collect the terms on documents and also their respective frequencies. This demanded a change in the pre-processing task in order to count the number of occurrences of each word. Until now, we discarded all duplicated words. Thus, the *terms* function in this strategy is defined as:

$$terms(document) = \{(term_1, frequency_1), \cdots, (term_n, frequency_n)\}, \qquad (4.2)$$

where $term_i$ is a word in document, and $frequency_i$ is the count of that word into the document. And the *sorted* function is defined as:

$$sorted(t) = (term_{k_1}, term_{k_2}, \cdots, term_{k_n}),$$

where,

$$\forall k_i, k_j (k_i \leq k_j \Rightarrow frequency_{k_i} \geq frequency_{k_j}),$$
$$t = \{(term_1, frequency_1), (term_2, frequency_2), \cdots, (term_n, frequency_n)\},$$
$$1 \leq k_i, k_j \leq n.$$

In fact, before doing the actual sorting, we modify the frequencies to decrease the relevance of common terms, i.e. terms which are often highly frequent in all clusters. To do that we subtract the average frequency of the term from the frequency of the term inside the cluster. This was suggested by Kuhn et al [51].

4.6.2 Second Strategy: LSI

Our second attempt was based on the method which Kuhn et al [51] used to label the clusters on their experiment. Their method is to use the LSI-index as a search engine. To do that, we first build a LSI-index containing term-term similarities, and then, we use the terms inside the documents as a query string and we submit that query to the LSI-index, which returns a set of terms with their respective indexes of similarity with regard to the query. Specifically,

$$terms(document) = \{(term_1, similarity_1), \cdots, (term_n, similarity_n)\} \qquad (4.3)$$

where *term* is a word from LSI-index, and *similarity* is the similarity index of this word in the context of the document. And the *sorted* function is defined as:

$$sorted(t) = (term_{k_1}, term_{k_2}, \cdots, term_{k_n}),$$

where,

$$\forall k_i, k_j (k_i \leq k_j \Rightarrow similarity_{k_i} \geq similarity_{k_j}),$$
$$t = \{(term_1, similarity_1), (term_2, similarity_2), \cdots, (term_n, similarity_n)\},$$
$$1 \leq k_i, k_j \leq n.$$

Again, before doing the actual sorting, we modify the similarities to decrease the relevance of common terms subtracting the average similarity of the term from the similarity of the term inside the cluster.

4.6.3 Results

In Table 4.4 and 4.4, we provide samples of terms extraced from a random fine-grained co-change cluster. Table 4.6 shows the entities associated with that cluster. As we can see, the cluster is associated with some kind of service which provides a API to verify *change requests* (*pedido de alteração* in portuguese). Here, we assume that *change request* is a expression well known among people familiar with the problem domain.

momento	70	recurso	67	web	59
servic	47	perfil	44	id	40
usuario	35	permissao	35	integ	24
credenci	21	log	21	name	21
exercicio	18	siop	18	dto	17
verificacao	17	retorno	16	servico	15
alteracao	13	pedido	13	permisso	12
tem	12	soap	11	param	11
list	11	erro	10	passou	8
detalh	8	obter	8	request	7

Table 4.4: Top 30 Terms Extracted Using First Strategy (With Frequency Numbers)

mous	0.44	node	0.40	sintes	0.40
first	0.40	interceptor	0.39	falha	0.39
estimada	0.38	anexo	0.38	identificadorid	0.38
precatorio	0.37	percent	0.37	qname	0.36
credenci	0.36	logoperacao	0.36	adciona	0.36
informado	0.36	classificaco	0.35	timestamp	0.35
push	0.35	atrasado	0.34	retorna	0.34
enabl	0.34	ali	0.37	csv	0.34
transposicao	0.33	selector	0.33	ordena	0.33
loc	0.32	decl	0.32	nometabela	0.32

Table 4.5: Top 30 Terms Extracted Using Second Strategy (With Similarity Numbers)

In Table 4.4, the most frequent terms are generic words (translated to english: *moment, service, user, credentials, year, verification*, etc.). Some terms associated with the problem domain (*request*, or *pedido* in portuguese) or with the technology (*soap*), appears on the list, but with lower frequency.

Class	Method
PermissaoServicoLocal	getMomentoUsuario(Perfil,Recurso)
	isTemPermissao(Perfil,Recurso,Momento)
	obterRecurso(Integer)
RetornoVerificacaoPedidoAlteracaoDTO	verificacoes
	getVerificacoes()
	setVerificacoes(List)
VerificacaoPedidoAlteracaoDTO	getDetalhes()
	getRegra()
	isPassou()
VerificacaoPedidoAlteracaoDTO	enviarPedidoAlteracao(CredencialDTO)

Table 4.6: Sample Entities Associated With the Cluster

In Table 4.5, there are some words related to problem domain (like *precatorio,classificaco*), but the majority of the words are generic or unrelated.

4.6.4 Discussion

i

As we saw in previous subsection, the terms extracted from clusters using the technique proposed provides support for concept discovery, but is not sufficient. To draw strong conclusions, a complete experiment must be made, but based on this small sample we can make some conjectures. The first strategy has a better performance, at least for our sample cluster. The problem with the two strategies is the outcome of generic words, despite the use of the suggestion of Kuhn et al [51], that reduces the importance of general frequent words.

We can imagine the use of stopwords to avoid generic words, but this can be a lot of effort, since we can have thousands of them, and, to do this, domain knowledge is required too. Nevertheless, we can also infer the problem domain concept which is associated with the cluster by examining the classes and method names contained in it. Thus, we believe that a fully automated approach to extract concepts from co-change clusters is not feasible. But we do believe that these lists combined can be useful to aid a developer in a software comprehension task in general, and, in particular, to discover where the concepts are implemented.

4.7 Implications of our results

In this section we discuss the implications of the main finding of this chapter: that is, fine-grained co-change clusters present a high degree of conceptual cohesion—where the notion of conceptual cohesion relates to the vocabulary present in source-code elements [59]. Therefore, the fine-grained co-change clusters provide a complementary view of a high cohesive modular decomposition. In particular, those clusters might help the identification of concepts that scatter throughout different classes. In that sense, the vocabulary which represents the semantic of a fine-grained cluster can be viewed as a kind of code annotation, that might allow *virtual separation of concerns*[47]. Also, the coverage of fine-grained clusters in relation to code entities can be expanded by applying specific techniques, such as the approach proposed by Nunes et al. [63]. This way, the mapping between the concepts embedded in the fine-grained clusters might serve as an initial *seed* to relate features and source-code. Additionally, each co-change cluster have a list of terms semantically related that can be used to infer the concepts associated to a particular cluster, or to search certain clusters associated with a query expressed as a list of correlated terms. Again, this can aid feature identification or location, and we envision its use in conjunction with search based tools for software maintenance.

4.8 Threats to Validity

In this section we present a discussion about some questions that might threat the validity of our work. We organize them according to the internal, construct, and external threats.

Internal Validity. We applied the same method to all target systems, including thresholds. However, we cannot ensure that some combination of thresholds favor or disfavor a particular project. Specifically, the effect of the threshold used to limit the maximum entities per commit depends on the development process of the subject project. To minimize these effects, we chose the thresholds according to the guidance of previous studies. When searching for the best combination of thresholds, in regard to semantic similarity, on the majority of projects the same combination was selected as the best. As the co-change clusters contain only a fraction of the original set of code entities, it is possible that certain portions of one project would produce metrics more favorable than of another project. But, as the results show a clear trend among most systems, for the computed metrics, probably the effect of this threat is not significant. The number of clusters generated by `Bunch` depend on its algorithm, and thus it can interfere on the metrics values as well. Our results showed that there is a strong inverse correlation between the clusters growth (influenced by the number of clusters), and their conceptual

cohesion (see Section 4.5). Thus, the effect of this threat is not significant, because this correlation is consistent in all systems.

Construct Validity. While we require an association between issues and commits to raise the semantic relation of the clusters, the precision of this association cannot be ensured. That is, we can not verify whether a given commit is related to the real issues that motivated the software revision. In addition, this association reduces the number of commits considered in our analysis. This reduction depends on the proportion of commits associated to issues in each system, as show in Section 4.4. Also, we have to constrain the history period when building the graphs for some projects, due to a `Bunch` limitation. But the potential impact of this threat does not invalidate our results, according to the analysis in Section 4.5. This analysis show that, if we analyze only the constrained period, the results will be even more favorable to the fine-grained co-change clusters. The representation of the graph was in conformance with existing works, albeit there are alternatives [18]. Our approach has some differences from the majority of studies. In particular, we use fine-grained entities as elements; and the issues in common as dependency criteria instead of commits.

External Validity. We selected a small set of *Java* projects for this study. This can potentially limit the generalization of our results. Nevertheless, these projects have been used in previous research works, and we choose a wide range of applications, not limited to open-source ones. All projects had large codebases with a long history of changes. As our purpose is to disclosure the conceptual cohesion of co-change clusters, we expect that the the methodology we present in this chapter can be reproduced in other projects. The small number of preserved entities in fine-grained clusters can threat the validity of the results for this level of granularity. Nevertheless, even this small set of entities can represent latent concepts inside codebases, and, as such, can be expanded to discover more code fragments related to the same concept [63], and, thus, raise its usefulness. We used `Bunch` as the only tool for software clustering. The choice for `Bunch` was made after a broad research on the literature, and it was found among the tools that produce the better results for software clustering [57]. Also, the `Bunch`'s limitations were mitigated according to the previous explanations.

4.9 Related Work

The work of Gall et al. [36] was the first to explore the information from version history repositories to detect evolutionary coupling between modules. Zimmermann et al. [79] proposed an approach to determine evolutionary coupling between fine-grained entities, used for predicting further changes [81]. Beyer and Noack [18] introduced the use of

co-change dependencies in clustering, while Vanya et al. [74] proposed a semi-automatic approach to suggest remodularization. In their approach, given an initial partition, inter-partitions evolutionary dependencies are identified. Silva et al. [71] propose an approach to assess modularity using co-change clusters. Their method use the Chameleon [46] tool to cluster coarse-grained entities that are compared with the actual package decompositions. According to their work, mismatches between the co-change clusters and the package decomposition can suggest new directions for restructuring the package hierarchy.

Regarding the semantic assessment of source-code, Maletic and Marcus [56] introduced the use of LSI to extract semantic information code entities. Marcus and Poshyvanyk [59] proposed new measures for class cohesion based on LSI. These measures, though conceptual, are correlated with traditional software cohesion measures. Their metric (*Conceptual Cohesion of Classes (C3)*) computes the average similarity index between each pair of methods of a class. For each method, a document is built, containing the terms extracted from identifiers and comments. Kuhn et al. [51], introduced *Semantic Clustering*, a technique based on LSI to group source-code artifacts that use the same vocabulary. To enable clustering, they use a similarity metric based on C3, generalizing it in order to compute the similarity of a cluster. In this case, the metric is defined as the average C3 of the classes contained in the cluster.

Bavota et al. [14] proposed the use of semantic information of the source-code, combined with its structural information to recommend remodularization. Santos et al. [69] experimented with semantic clustering also to evaluate software remodularizations. Their approach compares conceptual metrics values between a number of versions of a system, to analyze the relationship between the remodularizations promoted by the new versions and the semantic clusters. They also propose new metrics for conceptual cohesion of clusters and packages, that are the average cosine similarity between each pair of classes in clusters ans packages, respectively. Dit et al. [31] proposes an approach to measuring the textual coherence of the user comments in bug reports using Latent Semantic Analysis (LSA). They define the semantic similarity of a bug report as the average cosine similarity of the comments, and the coherence of a bug report as the semantic similarity. Silva et al. [71] use a similar approach, but they compute the similarity of the issues associated with the clusters using only the issues' short description, instead of the whole issue report.

This chapter is different from the aforementioned works in several aspects. First, our approach uses fine-grained entities and thus it increases the software cohesion of the resulting clusters, when compared to coarse-grained alternatives. In particular, we differ from the work of Kuhn et al. [51] because they compute the clusters based on the semantic similarity between classes. Thus, there are two main differences in this case: we compute the clusters based on the co-change dependencies; and we use finer-grained source-code

entities instead of only classes as entities. Our intention is to verify the conceptual cohesion of co-change clusters, so we are not interested in proposing a new clustering method. Also, the use of fine-grained entities allows us to discover interesting conceptual properties of the co-change clusters based on them, with results much more significant than for the coarse-grained clusters. In relation to the work of Marcus and Poshyvanyk [59], our work uses an extended version of their metrics. Although their work concentrates on analyzing the merit of the proposed metric (C3), our work extends the original definition and we do not compare its performance with another equivalent cohesion metric.

4.10 Conclusion

Software clustering use coupling information involving code entities to group entities with strong dependencies. Recent works have experiment with different kinds of software clustering techniques, including structural, dynamic, semantic, and based on change history; also with different levels of code entities granularity, but particularly with coarse-grained ones (like source-code files, C++ namespaces, Java packages, and object-oriented classes). In this chapter we presented a novel perspective of the software decomposition, based on fine-grained co-change clusters. We also investigated the conceptual cohesion of the resulting decomposition perspective, using a well-known measure of semantic similarity. Our evaluation considered seven real target systems, six open-source projects and one system from the financial domain of the Brazilian Government. The main conclusion is that *fine-grained co-change clusters* present a higher degree of conceptual cohesion when compared to the typical decomposition of the systems (based on the package hierarchy) and to another perspective based on *coarse-grained co-change clusters*. Therefore, our new perspective of the software decomposition, though limited in the number of entities, present a cohesive vocabulary associated to each cluster, allowing the discovery of concepts scattered on a number of code entities. As a future works, we aim at investigating the hypothesis that combining our perspective with existing techniques for feature expansion would help on the identification of features during reverse engineering activities.

Chapter 5

Conclusion

In this work we explored the properties of the software entities clustering which are evolutionarily coupled, with regard to its suitability to represent high level concepts from the domain of a software system. The research results revealed that co-change clusters by itself does not have good modularity properties, and are not a good model to follow when the intention is to restructure an architecture of a software system. For the other side, the research revealed that fine-grained co-change clusters have superior conceptual cohesion when compared to both coarse-grained co-change clusters and modular units. Thus, we made some exploratory analysis trying to answer the following question: *the terms associated with fine-grained co-change clusters help to identify concepts from the problem domain?*

The ultimate answer to this question can be very difficult to be found, but we can extract the terms associated with the co-change clusters, according to some method, get some domain experts to evaluate the outcome, and analyze if they make sense and are useful. While this task was not possible to be executed given our time frame, we performed some exploration of the available methods for terms extraction and analyzed preliminarily their performances. The results shown that a fully automated procedure to recover problem domain concepts from source-code artifact is not feasible, but the extracted term can be useful when used to find the implementation elements of some concept. We also believe that the terms extracted from several methods, when combined, can aid the domain experts to infer the problem domain concept related to them.

5.1 Summary of the Contributions

The main contributions of this work are:

- An approach for reasoning about the hidden dependencies induced by the co-evolution of software assets that is supported by a general theory of modularity. Thus, we

propose the use of a well defined set of tools and metrics for reasoning about modularity.

- A comprehensive study about the impact of co-evolution with respect to the design structure of one proprietary and six Java open-source softwares that we use as target systems.
- We describe a framework for building a different perspective of a software (based on fine-grained co-change clusters) and a methodology to evaluate the conceptual cohesion of this software perspective.
- We report the results of an empirical assessment of our software perspective. In this analysis, we compute fine-grained co-change clusters for seven real-world systems and the observations reveal a higher cohesion of the resulting abstractions with respect to the typical decomposition of the systems.

Besides these contributions, we also made early evaluations of terms extraction from the co-change clusters, but a more comprehensive study must be made. It would also be useful to design and run a experiment aiming to asses if these terms can be used to identify features. However, because the time limitation of this work, these experiments will be done later.

5.2 Impact on the Organization

Currently, some results of this dissertation are being useful in a software modernization effort at the Ministry of Planning, Budget and Management (MP). For instance, within the scope of this project, we use the clustering approach discussed in Chapter 3 for helping us to:

- Identify focus of modularity issues, such as cyclical dependencies and frequently mutual changes in different modules leading to undesirable evolutionarily dependencies.
- Suggest a new set of modules and modules' contents, that is, to use coarse-grained clusters as a reference for remodularization, which can be useful for finding ideas for new modules. As shown in Chapter 3, it is not recommended to follow this strategy indiscriminately, but it can be used for finding specific recommendations.
- Compare SIOP's architecture with other systems that are more resilient to changes and to learn from them. As discussed in Chapter 3, systems such as Hadoop have more modular architectures and can be used as an inspiration for the new modular structure for SIOP.

With regard to SIOP features identification, though not fully completed yet, we have found a promising approach that will be further investigated and refined in that direction, namely the fine-grained co-change clusters. As we discussed in Chapter 4, the conceptual cohesion of these clusters are superior to the coarse-grained clusters and the actual classes and interfaces. Accordingly, the next step is to use these fine-grained clusters computed for SIOP as seeds for a feature expansion tool and to analyze its output considering two aspects: (a) *the expanded clusters maintain a higher conceptual cohesion?* and (b) *the expanded clusters are identifiable with SIOP features?*

Clearly, if successful, these further investigations will benefit the MP and SIOP, as we will have a list of features identified and associated with the corresponding source-code. Nevertheless, these results can benefit several other systems too, as the goal, methodology, and approach are general and use available tools.

5.3 Future Work

The clustering technique applied to fine-grained software entities, as our findings revealed, have a strong potential for future research. In the following subsection, we will discuss further developments which we envision.

5.3.1 Providing Seeds for Feature Expansion

While the conceptual cohesion of fine-grained co-change clusters are promising, their low coverage of source-code entities (4.21% on average), as we have seen in Chapter 4, is a barrier when we aim to identify features. Thus, we can research ways to raise their coverage. One of the alternatives is to provide the fine-grained clusters as input to feature expansion tools [63]. These tools start from a feature list and some mappings from features to source-code entities, and, based on heuristics, expand the mappings using static and evolutionary data.

5.3.2 Feature Location

The comprehension of software source-code is among the main concerns of software engineering research. The fine-grained co-change clusters can be used to aid the location of concepts scattered in source-code, and, in particular, the concepts which refer to features. We can explore the implementation of search tools that are based on the semantic indexes built for computation of conceptual cohesion. These tools can provide an user interface that allow to submit queries built from search terms, and can show the results linked with the source-code of the associated entities. The quality of the indexes must be researched

too, especially in regard to the relevance of the results when combining several sources of terms which are associated with entities.

5.3.3 Remodularization

While the co-change clusters are not full suitable to become modules, as we have seen in Chapter 3, the combination of the evolutionary coupling measure with other kinds of coupling can have potential. Some recent works has begun exploring the combination of coupling metrics, including evolutionary, both in conjunction with software clustering algorithms, such as [16], and using search based approaches [61]. While the remodularization research has solid contributions, several research questions are open yet.

Bibliography

[1] Dataset presented on Oliveira et al. Paper. http://goo.gl/3E761S. 22

[2] DependencyFinder tool. http://depfind.sourceforge.net/. 26

[3] GitHub source code repository supporting Oliveira et al. Paper. http://github.com/mcesarhm/mpca. 22

[4] Siop: Citizen service letter. http://www.orcamentofederal.gov.br/biblioteca/cartas-de-servico/carta_de_servicos_SIOP.pdf. 50

[5] Nicolas Anquetil and Timothy C. Lethbridge. Experiments with clustering as a software remodularization method. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 235–255. IEEE. 13, 38

[6] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009. 2, 9, 10, 11

[7] Sven Apel, Christian Lengauer, Don Batory, Bernhard Möller, and Christian Kästner. An algebra for feature-oriented software development. *University of Passau, MIP-0706*, 2007. 9

[8] Carliss Baldwin, Alan MacCormack, and John Rusnak. Hidden structure: Using network methods to map system architecture. *Research Policy*, 43(8):1381–1397, 2014. 7

[9] Carliss Young Baldwin and Kim B Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000. 5, 6, 18, 38, 39

[10] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Professional, 2013. 6

[11] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *Software Engineering, IEEE Transactions on*, 30(6):355–371, June 2004. 2

[12] Don Batory. Feature modularity for product-lines. *Tutorial at: OOPSLA*, 6, 2006. 9

[13] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, 49(12):45–47, 2006. 9

[14] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):4, 2014. 64

[15] Fabian Beck and Stephan Diehl. Evaluating the impact of software evolution on software clustering. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 99–108. IEEE, 2010. 52

[16] Fabian Beck and Stephan Diehl. On the impact of software evolution on software clustering. *Empirical Software Engineering*, 18(5):970–1004, 2013. 13, 14, 16, 25, 26, 27, 28, 29, 39, 40, 45, 47, 50, 51, 56, 69

[17] Fabian Beck and Stephan Diehl. Visual comparison of software architectures. *Information Visualization*, 12(2):178–199, 2013. 38

[18] Dirk Beyer and Andreas Noack. Clustering software artifacts based on frequent common changes. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 259–268. IEEE, 2005. 37, 38, 63

[19] Dirk Beyer and Andreas Noack. Mining co-change clusters from version repositories. Technical report, 2005. 14, 42

[20] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach.* Pearson Education, 2000. 9

[21] Pierre Bourque, Richard E Fairley, et al. *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0.* IEEE Computer Society Press, 2014. 3

[22] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. An approach to constructing feature models based on requirements clustering. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 31–40. IEEE, 2005. 9

[23] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. In *Fundamental Approaches to Software Engineering*, pages 16–30. Springer, 2008. 9

[24] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond.* Pearson Education, 2010. 6

[25] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*, volume 59. Addison-Wesley Reading, 2002. 10

[26] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. 9

[27] Scott C. Deerwester, Susan T Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *JAsIs*, 41(6):391–407, 1990. 43, 48

[28] Premkumar Devanbu, Bob Balzer, Don Batory, Gregor Kiczales, John Launchbury, David Parnas, and Peri Tarr. Modularity in the new millenium: A panel summary. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 723–724, Washington, DC, USA, 2003. IEEE Computer Society. 1, 7

[29] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 341–350. IEEE, 2015. 45

[30] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. *A discipline of programming*, volume 4. prentice-hall Englewood Cliffs, 1976. 6

[31] Bogdan Dit, Denys Poshyvanyk, and Andrian Marcus. Measuring the semantic similarity of comments in bug reports. *Proc. of 1st STSM*, 8, 2008. 64

[32] Susan T Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers*, 23(2):229–236, 1991. 48

[33] Steven D. Eppinger and Tyson R. Browning. *Design Structure Matrix Methods and Applications*. MIT Press, 2012. 19

[34] Steven D Eppinger, Daniel E Whitney, Robert P Smith, and David A Gebala. A model-based method for organizing tasks in product development. *Research in Engineering Design*, 6(1):1–13, 1994. 8

[35] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 13–23, Sept 2003. 14, 42

[36] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 190–198. IEEE, 1998. 38, 63

[37] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 486–496. IEEE, 2013. 14

[38] Markus M Geipel and Frank Schweitzer. The link between dependency and cochange: Empirical evidence. *Software Engineering, IEEE Transactions on*, 38(6):1432–1444, 2012. 40

[39] A.E. Hassan and R.C. Holt. Predicting change propagation in software systems. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 284–293, Sept 2004. 42

[40] Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008. 12

[41] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Historage: fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 96–100. ACM, 2011. 23, 44

[42] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 121–130. IEEE, 2013. 37, 45

[43] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007. 12

[44] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990. 9, 10

[45] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-; oriented reuse method with domain-; specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998. 9

[46] George Karypis, Eui-Hong Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999. 39, 64

[47] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, pages 311–320. ACM, 2008. 62

[48] Mik Kersten and Gail C. Murphy. Mylar: A degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 159–168, New York, NY, USA, 2005. ACM. 1, 7

[49] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 1–11, New York, NY, USA, 2006. ACM. 42

[50] Miryung Kim and David Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM, 2005. 13

[51] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007. 42, 48, 50, 59, 61, 64

[52] Matthew J LaMantia, Yuanfang Cai, Alan D MacCormack, and John Rusnak. Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases. In *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*, pages 83–92. IEEE, 2008. 38

[53] Cristina Videira Lopes and Sushil Krishna Bajracharya. An analysis of modularity in aspect oriented design. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26. ACM, 2005. 18

[54] Cristina Videira Lopes and Sushil Krishna Bajracharya. Assessing aspect modularizations using design structure matrix and net option value. In *Transactions on Aspect-Oriented Software Development I*, pages 1–35. Springer, 2006. 18

[55] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Manage. Sci.*, 52(7):1015–1030, July 2006. 9, 18, 19, 21, 27, 33, 35, 36

[56] Jonathan I Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112. IEEE Computer Society, 2001. 42, 50, 64

[57] Onaiza Maqbool and Haroon Babri. Hierarchical clustering for software architecture recovery. *Software Engineering, IEEE Transactions on*, 33(11):759–780, 2007. 38, 63

[58] Onaiza Maqbool and Haroon A Babri. Hierarchical clustering for software architecture recovery. *Software Engineering, IEEE Transactions on*, 33(11):759–780, 2007. 14, 38

[59] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 133–142. IEEE, 2005. 42, 44, 48, 49, 50, 62, 64, 65

[60] Brian S Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *Software Engineering, IEEE Transactions on*, 32(3):193–208, 2006. 13, 25, 47

[61] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):17, 2015. 69

[62] Gail C Murphy, Mik Kersten, Martin P Robillard, and Davor Čubranić. The emergent structure of development tasks. In *ECOOP 2005-Object-Oriented Programming*, pages 33–48. Springer, 2005. 17, 30, 42

[63] Camila Nunes, Alessandro Garcia, Carlos Lucena, and Jaejoon Lee. Heuristic expansion of feature mappings in evolving program families. *Software: Practice and Experience*, 2013. 43, 62, 63, 68

[64] Marcos Oliveira, Rodrigo Bonifacio, Guilherme Ramos, and Marcio Ribeiro. On the conceptual cohesion of co-change clusters. In *CBSoft 2015 - SBES 2015 - Technical Research ()*, Belo Horizonte, sep 2015. 4

[65] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972. 1, 6, 16

[66] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media, 2005. 3

[67] Kata Praditwong, Mark Harman, and Xin Yao. Software module clustering as a multi-objective search problem. *Software Engineering, IEEE Transactions on*, 37(2):264–282, 2011. 14

[68] G. Santos, K. Santos, Marco Tulio Valente, D. Serey, and Nicolas Anquetil. Topicviewer: Evaluating remodularizations using semantic clustering. In *IV Congresso Brasileiro de Software: Teoria e Prática (Sessao de Ferramentas)*, pages 1–6, 2013. 38

[69] Gustavo Santos, Marco Tulio Valente, and Nicolas Anquetil. Remodularization analysis using semantic clustering. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 224–233. IEEE, 2014. 38, 42, 50, 64

[70] Luciana Silva, Marco Tulio Valente, and Marcelo Maia. Co-change clusters: Extraction and application on assessing software modularity. In *Transactions on Aspect-Oriented Software Development*, pages 1–37, 2015. 16, 17, 25, 28, 33, 37, 39

[71] Luciana Lourdes Silva, Marco Tulio Valente, and Marcelo de A. Maia. Assessing modularity using co-change clusters. In *Proceedings of the of the 13th international conference on Modularity*, pages 49–60. ACM, 2014. 42, 45, 51, 64

[72] Donald V Steward. The design structure system: a method for managing the design of complex systems. *IEEE transactions on Engineering Management*, (EM-28), 1981. xi, 7, 8

[73] Kevin J Sullivan, William G Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering Notes*, 26(5):99–108, 2001. 18

[74] Adam Vanya, Lennart Hofland, Steven Klusener, Piërre Van De Laar, and Hans Van Vliet. Assessing software archives with evolutionary clusters. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 192–201. IEEE, 2008. 39, 64

[75] Theo A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 33–43. IEEE, 1997. 13, 38, 44

[76] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971. 6

[77] Lu Xiao, Yuanfang Cai, and Rick Kazman. Design rule spaces: a new form of architecture insight. pages 967–977. ACM Press. 38

[78] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 73–83. IEEE. 15

[79] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 73–83. IEEE, 2003. 16, 17, 25, 38, 39, 45, 63

[80] Thomas Zimmermann and Peter Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6. sn, 2004. 42

[81] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005. 13, 25, 38, 42, 45, 63