

**ANÁLISE DE REMODULARIZAÇÃO USANDO
AGRUPAMENTO SEMÂNTICO**

GUSTAVO JANSEN DE SOUZA SANTOS

**ANÁLISE DE REMODULARIZAÇÃO USANDO
AGRUPAMENTO SEMÂNTICO**

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

ORIENTADOR: MARCO TULIO VALENTE

Belo Horizonte

Janeiro de 2014

GUSTAVO JANSEN DE SOUZA SANTOS

**REMODULARIZATION ANALYSIS USING
SEMANTIC CLUSTERING**

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: MARCO TULIO VALENTE

Belo Horizonte

January 2014

© 2014, Gustavo Jansen de Souza Santos.
Todos os direitos reservados.

Santos, Gustavo Jansen de Souza

S237r Remodularization Analysis Using Semantic
Clustering / Gustavo Jansen de Souza Santos. — Belo
Horizonte, 2014
xxiv, 70 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de
Minas Gerais. Departamento de Ciência da Computação
Orientador: Marco Tulio Valente

1. Computação – Teses. 2. Engenharia de Software –
Teses. 3. Software – Reutilização – Teses. I. Orientador.
II. Título.

CDU 519.6*32(043)



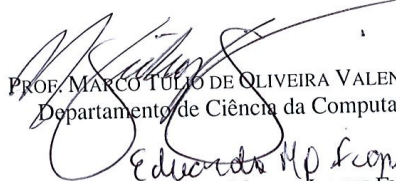
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


FOLHA DE APROVAÇÃO

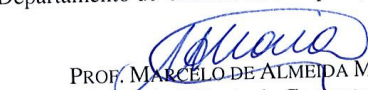
Análise de modularização usando agrupamento semântico

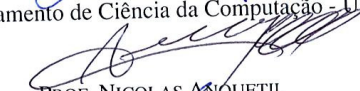
GUSTAVO JANSEN DE SOUZA SANTOS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


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


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


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


PROF. NICOLAS ANQUETIL
Universidade de Lille-1 - França


PROF. RICARDO TERRA NUNES BUENO VILLELA
Departamento de Tecnologia em Engenharia Civil, Computação e Humanidades - UFSJ

Belo Horizonte, 23 de janeiro de 2014.

To all the people—and songs—that helped me through this path.

Acknowledgments

Two years that were a lifetime. I would like to thank my family who supported me in this “crazy” decision to move out. Also to the friends that I made in this period. Special thanks to my old friends that kept me sane with the most insane conversations about life.

I also would like to thank Marco Tulio Valente for the support as educator, advisor, and fellow researcher. Thanks to Nicolas Anquetil for proposing this work and also for the opportunity of working as intern with the RMoD Team.

Thanks to my colleagues from ASERG and RMoD teams for the company, nice talks, and feedback to this work.

I thank the secretariat of DCC for all the support, especially in these last weeks. Thanks CNPq for financial support.

“Stop listening to the static.”
(Nathaniel Samuel Fisher, Jr.)

Resumo

Durante a evolução de software, sua estrutura inevitavelmente se torna mais difícil de manter, a menos que um esforço explícito de manutenção seja feito para melhorá-la. No intuito de resolver esse problema, a recomendação comum consiste em organizar a estrutura do software em módulos. Esse processo é geralmente realizado através da otimização de valores de métricas estruturais de coesão e acoplamento, por exemplo. No entanto, trabalhos recentes começam a questionar a utilidade de métricas estruturais. Mais especificamente, essas métricas não expressam integralmente a melhoria na arquitetura resultante de um processo de manutenção. Nesta dissertação de mestrado, é seguida metodologia existente para avaliar métricas de software considerando remodelarizações reais, ou seja, cuja manutenção foi realizada pelos arquitetos do software. Foi utilizado um conjunto de métricas que consideram similaridade textual entre artefatos de software, chamadas métricas conceituais. Para realizar essa tarefa, foi relatado um experimento sobre o uso de Agrupamento Semântico (*Semantic Clustering*) para avaliar remodelarizações de software. Agrupamento Semântico é uma abordagem que se baseia em recuperação de informação e técnicas de agrupamento para extrair conjuntos de classes similares de acordo com seus vocabulários. Foi reportada a adaptação que realizou-se nessa técnica e esta adaptação foi avaliada usando seis remodelarizações de quatro sistemas de software. Observou-se que Agrupamento Semântico e métricas conceituais podem ser usados para expressar e explicar a intenção dos arquitetos ao realizar operações de modularização recorrentes, como decomposição de módulos.

Palavras-chave: Arquitetura de Software, Manutenção de Software, Remodularização, Processamento de Texto, Recuperação de Informação, *Semantic Clustering*, Métricas Conceituais.

Abstract

As software evolves, its structure inevitably gets harder to understand and maintain, unless explicit effort is done to improve it. To tackle this problem, the common recommendation consists in organizing the software structure into modules. This process is often performed by optimizing the value of cohesion and coupling metrics, for example. However, recent work question the usefulness of structural metrics. More specifically, structural metrics do not seem to fully express the architectural improvement resulted from software maintenance. In this master dissertation, we follow an existing methodology to assess software metrics regarding real remodularization cases, i.e., in which the maintenance was performed by the software's architects. We use a set of recently proposed metrics, called conceptual metrics, which consider textual similarity between software artifacts. To accomplish this task, we report an experiment on using Semantic Clustering to evaluate software remodularizations. Semantic Clustering is an approach that relies on information retrieval and clustering techniques to extract sets of similar classes in a system according to their vocabularies. In fact, we adapted Semantic Clustering to support remodularization analysis. We then evaluate our adaptation using six real-world remodularizations of four software systems. As a result, we conclude that Semantic Clustering and conceptual metrics can be used to express and explain the intention of architects when performing common modularization operations, such as module decomposition.

Keywords: Software Architecture, Software Maintenance, Remodularization, Text Processing, Information Retrieval, Semantic Clustering, Conceptual Metrics.

List of Figures

1.1	Overview of the proposed approach	5
2.1	Distribution Map of Colt	12
2.2	Components of an Information Retrieval System	13
2.3	Text extraction of the identifiers of class <code>SparseDoubleMatrix2D</code>	15
2.4	Tokenization of the text in <code>SparseDoubleMatrix2D</code>	15
2.5	Stemming of text in <code>SparseDoubleMatrix2D</code>	16
2.6	Semantic Clustering overview, extracted from Kuhn et al. [2007].	22
2.7	Example of a Distribution Map	26
3.1	Semantic Clusters Generation Algorithm	33
3.2	Distribution Maps in the remodularization of JHotDraw	34
3.3	TopicViewer's User Interface	36
3.4	TopicViewer's Architecture	37
3.5	Distribution Map visualization in Hapax 2.0	38
4.1	Number of clusters (a) and Internal Cohesion (b) of clusters for similarity thresholds ranging from 0.55 to 0.75 (discrete intervals of 0.05)	44
4.2	Spread results (the spread of the concepts <i>above</i> the diagonal increased after the remodularization)	47
4.3	Focus results (the focus of the concepts <i>above</i> the diagonal increased after the remodularization)	48
4.4	Conceptual Cohesion results of restructured packages (a) and new packages (b). The cohesion of the packages <i>above</i> the diagonal line increased after the remodularization	55

List of Tables

3.1	Regular Expressions for Java Source Code Filtering	31
3.2	Additional Stopwords	31
4.1	Vocabulary Data (NOC= number of classes; NOP= number of packages) .	41
4.2	Threshold Selection (Clu= # clusters)	45
4.3	Wilcoxon test results for spread and focus of clusters	49
4.4	Modularization operators responsible for the Top-3 and the Bottom-3 changes in spread and focus (MD= module decomposition; MU= module Union; FT= file transferal; DT= data structure transferal; RN= rename; PF= promote function; MR= module removal; FR= file removal)	51

Contents

Acknowledgments	xi
Resumo	xv
Abstract	xvii
List of Figures	xix
List of Tables	xxi
1 Introduction	1
1.1 Problem Description	2
1.2 Goals and Contributions	3
1.3 Organization	4
2 Background	7
2.1 Software Remodularization	7
2.1.1 Modularization Operators	9
2.2 Software Architecture Reconstruction	10
2.3 Distribution Map	11
2.4 Information Retrieval	12
2.4.1 Latent Semantic Indexing	18
2.4.2 Information Retrieval Techniques in Reverse Engineering	20
2.5 Semantic Clustering	21
2.6 Structural Metrics	23
2.7 Conceptual Metrics	24
2.8 Final Remarks	27
3 Improvements to Semantic Clustering	29
3.1 Text Filtering Strategies	29

3.2	Semantic Clustering Stop Criterion	31
3.3	Semantic Clusters Generation	32
3.3.1	Visualization Support	33
3.4	Tool Support	34
3.4.1	Topic Viewer	34
3.4.2	Hapax 2.0	37
3.5	Final Remarks	38
4	Evaluation	39
4.1	Target Systems	39
4.2	Methodology	40
4.3	First Study: Semantic Clustering Setup	42
4.4	Second Study: Remodularization Analysis	46
4.4.1	Impact of Remodularizations on Semantic Clusters	46
4.4.2	Remodularization Operators with the Highest Impact in Semantic Clusters	49
4.4.3	Impacts of Module Decomposition in Conceptual Cohesion	52
4.5	Threats to Validity	54
4.6	Final Remarks	57
5	Conclusion	59
5.1	Contributions	59
5.2	Publications	60
5.3	Future Work	60
	Bibliography	63

Chapter 1

Introduction

A critical aspect in the design of any complex software system is its architecture, i.e., its high-level organization as a collection of components and the interactions between them [Garlan and Perry, 1995]. The architecture of a system prescribes the organization and interaction of its components, and the principles that guide the design of the system over time [Garlan, 2000]. However, as a software evolves, its structure inevitably gets more complex and harder to understand. Modifications become more difficult to implement, unless explicit effort is done to improve the architecture [Lehman, 1996]. Remodularization consists in a sequence of modifications restricted to the structure of the software. Remodularization is then recommended to improve the original architecture by organizing the system's components in a modular structure [Anquetil and Laval, 2011].

Despite the experience and good practices in software maintenance, there is no solid agreement on what constitutes a good architecture [Anquetil and Laval, 2011]. The perception of architectural quality is subjective and also of great importance. A grounded definition of architectural quality would assist the developers on the design and the evaluation of candidate architectures during software maintenance and remodularization. Metrics and tools would be created in order to assist developers and architects in achieving a better design for their software.

In most cases, remodularizations are guided by structural aspects, i.e., the static dependencies between architectural components such as method calls or attribute usages [Ujhazi et al., 2010; Anquetil and Laval, 2011]. For example, a common recommendation is that (structural) cohesion should be maximized and coupling should be minimized [Stevens et al., 1974]. In other words, a good architecture is the one whose components have more dependencies between its internal elements (i.e., the components are cohesive) and less dependencies between elements of other components (i.e.,

the components have low coupling). This recommendation can be manually followed or with the help of metrics and refactoring tools [Mancoridis et al., 1999].

On the other hand, recent research proposes the use of lexical information from source code [Corazza et al., 2011]. This information is often applied to concept location and program comprehension. This lexical information, described for example in identifiers and comments, can express the intention of developers [Kuhn et al., 2007; Abebe et al., 2009]. Such approach is promising since it is estimated that up to 70% of the source code (in number of characters) consists of identifiers [Abebe et al., 2009]. Recent work also proposes cohesion and coupling metrics, based on textual similarity of software entities such as classes and methods [Marcus et al., 2004; Ujhazi et al., 2010; da Silva et al., 2012].

1.1 Problem Description

After decades in software measurement, a variety of metrics have been proposed in the literature. However, there is little evaluation of these metrics [Briand et al., 1998]. For example, recent work question the structural cohesion/coupling dogma, stating that “coupling and cohesion do not seem to be the dominant driving forces when it comes to modularization” [Abreu and Goulão, 2001; Anquetil and Laval, 2011]. In other work, Taube-Schock et al. [2011] stated that structural coupling metrics follow a power-law distribution and, therefore, components with very high coupling are inevitable. Another work showed that structural cohesion metrics usually present divergent results when used to evaluate the same refactoring actions [Ó Cinnéide et al., 2012].

Anquetil and Laval [2011] conducted an empirical study with structural metrics of cohesion and coupling, and real remodularization cases. This study contradicted the cohesion/coupling dogma, by observing an increase in coupling after the remodularizations. Therefore, it raises doubts about the use of traditional metrics to assist software maintenance. More specifically, conventional cohesion and coupling metrics do not fully represent the particular concepts behind them, as stated (in particular for cohesion) by Ó Cinnéide et al. [2012]; or the cohesion/coupling dogma is not sufficient to assess architectural quality, as stated by Abreu and Goulão [2001].

The study conducted by Anquetil and Laval [2011] motivated this master thesis since it proposes the evaluation of quality metrics in real remodularization cases. It provided insight on how software maintenance, as performed by developers, does not follow the cohesion/coupling dogma, at least as expressed by structural metrics. This fact needs attention because previous work relied on this dogma to propose new re-

modularization by improving structural cohesion and coupling [Mancoridis et al., 1999; Mitchell and Mancoridis, 2001].

Anquetil and Laval also discuss that other properties are overlooked by this type of metrics. For example, an aspect of cohesion might be the degree that the components of a software implement the same purpose [Sindhgatta and Pooloth, 2007; Kuhn et al., 2007; da Silva et al., 2012]. Recently proposed metrics capture this property by calculating the similarity of the text described in the source code. However, to the best of our knowledge, there is also little evaluation work in the literature on conceptual metrics applied to software maintenance. In a recent work, Bavota et al. [2013a] stated that semantic information, as captured by conceptual metrics, are more likely to express the developers' perception of identifying coupling between classes. Their work is one of a few that analyzes the relevance of quality metrics, such as coupling, according to the point of view of developers.

1.2 Goals and Contributions

The main goal of this master dissertation is to investigate the approach proposed by Anquetil and Laval [2011] for evaluating real remodularization cases and structural metrics. Basically, we extend this approach by considering conceptual metrics, i.e., metrics that analyze the information embedded in the comments and identifiers in software [Ujhazi et al., 2010]. Conceptual metrics do not consider the hierarchy of software artifacts nor the similarity of code fragments, unlike concern-based metrics [da Silva et al., 2012]. The similarity of software artifacts is computed only on the textual information provided by developers.

The first conceptual metric, *Conceptual Cohesion of a Package*, calculates the average similarity of all classes in a given package. The other two metrics we use in this dissertation rely on the notion of *concepts*, i.e., groups of entities that share the same purpose in the system [Ducasse et al., 2006; Kuhn et al., 2007]. Thus, *Spread* measures the number of packages which contains a given concept. On the other hand, *Focus* measures the concentration of a concept in the packages that it appears. In order to extract these concepts, we use Semantic Clustering, an approach that relies on information retrieval and clustering techniques to extract groups of classes with similar vocabulary [Kuhn et al., 2005, 2007].

Figure 1.1 presents an overview of our approach. Following the methodology proposed by [Anquetil and Laval, 2011], given a remodularization case, i.e., two versions of a system with an explicit remodularization effort, we execute Semantic Clustering

in order to extract the concepts in each version (Semantic Clustering process in Figure 1.1). Then, we compute the conceptual metrics for both versions and we check whether the metric values improved after the modularization (Conceptual Metrics Measurement step).

Regarding modularization analysis, we analyze the occurrence of basic modularization operators by correlating major changes in conceptual metric values to the operators that were applied (Modularization Operators step). Previous work categorize the operations more likely to occur in modularizations [Rama and Patel, 2010]. For example, modularization operators may include the decomposition of modules, the transferral of entities such as files and operations, and the creation of a module from smaller ones.

The study proposed in this master dissertation has the distinguishing contribution of analyzing how conceptual metrics evolve after an explicit modularization effort. This evaluation is not based on the perception of developers over a metric [Bavota et al., 2013a], but rather on how these metrics reveal this perception when improving the architecture. Moreover, we analyze the consequences of applying basic modularization operators according to conceptual metrics.

We also propose the adaptation of Semantic Clustering to support the comparison of two versions of a system. We force the number of generated clusters to be the same in both versions, in order to facilitate their comparison by means of conceptual metrics. Moreover, we propose the visualization analysis, using Distribution Maps [Ducasse et al., 2006], to verify the evolution of the clusters after the modularization. Finally, we present a tool that supports our adapted methodology.

1.3 Organization

This master dissertation is organized in four chapters, which are described as follows:

- Chapter 2 presents related work to the central theme of this master dissertation. More specifically, this chapter addresses the main goals of modularizations, provides an introduction on information retrieval applied to software engineering, and discusses the evaluation of metrics that are applied to software architecture. We also present Semantic Clustering, a visualization approach that extracts sets of similar classes in a system according to their textual information [Kuhn et al., 2007].

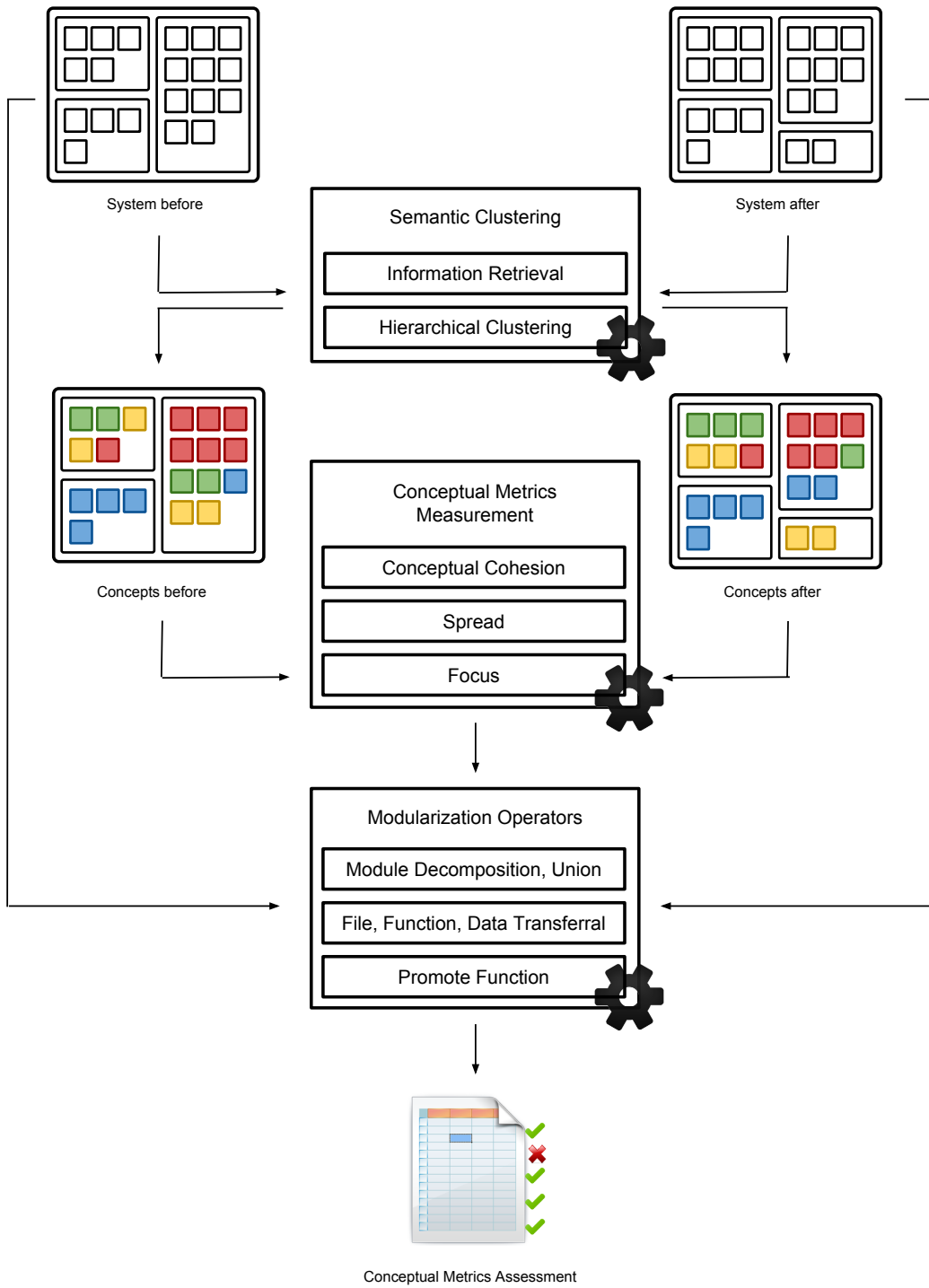


Figure 1.1. Overview of the proposed approach

- Chapter 3 presents the improvements we proposed to Semantic Clustering, in order to extract a feasible number of clusters and to support remodularization analysis. This chapter also presents the tool support of our approach.
- Chapter 4 presents the evaluation we conducted with conceptual metrics and real remodularization cases (Eclipse, JHotDraw, NextFramework and Vivo). We discuss the most common remodularization operators and whether conceptual metrics are able to express the architectural improvement in the systems under analysis.
- Chapter 5 concludes this dissertation and presents the contributions of the work. We also discuss the limitations of the proposed approach and directions for future research.

Chapter 2

Background

This chapter presents the related work to this master dissertation. Section 2.1 presents the definition of remodularization and challenges in this area. Section 2.1.1 reports recent work on categorizing the operators most likely to occur in remodularizations. Section 2.2 and Section 2.3 presents recent work in Software Architecture Reconstruction. Section 2.4 presents the basis of Information Retrieval techniques, as well as its application in Reverse Engineering; and Section 2.5 presents Semantic Clustering as the main technique that was applied in this dissertation. Section 2.6 and Section 2.7 reports recent work in evaluation of structural and conceptual metrics, respectively. Section 2.8 presents final remarks of the literature review.

2.1 Software Remodularization

Refactoring, as proposed by Fowler et al. [1999], is defined as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”. Basically, it is performed as a series of small changes that modify the *structure* of the code and *improve* it for further modifications. On the other hand, the definition also emphasizes that the new software must have the same functions as before. Therefore, refactoring is not applied to correct faults, or to add new features.

Although the definition explicitly cites maintainability as the main goal of refactoring, it is also used to achieve other goals. Demeyer et al. [2002] list other factors, as organizing a former monolithic system into modules, porting a system to a new platform, or supporting a new technology. In general, refactoring is a means to modify the code in order to provide some aspect in quality that it did not provide before (comprehensibility or portability, for example) [Fowler et al., 1999].

Fowler et al. [1999] also introduce the definition of Big Refactorings, also referred as Large Scale Refactorings, Architectural Refactorings or Remodularization in the literature. Big Refactoring is a major change in design and implementation that impacts the architecture. It is referred as “re”-engineering by the fact that one is rewriting parts of a system without changing its behavior. Only the design and implementation is changed [Bird, 2012].

Prior to re-design the existing code, it is necessary to understand how the design decisions are actually reflected in the source code. Architecture conformance techniques aim to discover the gap between concrete and planned architectures [Knodel and Popescu, 2007; Passos et al., 2010]. The motivation of such analysis rely on the fact that software architecture documentation is seldom maintained, if available at all [Bayer, 2004]. The main goal of architecture conformance is to prevent the accumulation of deviations from planned architecture by source code, a phenomenon known as architectural erosion [Perry and Wolf, 1992].

Knodel and Popescu [2007]; Passos et al. [2010] conducted a literature review on conformance checking techniques. These techniques include the extraction of static dependencies through search queries [Verbaere et al., 2008], checking architectural deviations by defining design rules [Murphy et al., 1995; Terra and Valente, 2009], or by mining design rules from software repositories [Maffort et al., 2013].

Thus, remodularizations are applied to improve the architecture in order to conform to the implemented design [Anquetil and Laval, 2011; Terra et al., 2014]. It is worth noting that remodularizations are not necessarily applied in the context of architectural erosion only, but also to improve other properties in the architecture, such as adaptation, portability, or evolution. The benefits of remodularization relates to software architecture benefits itself. New components can be built from well-defined interfaces and other projects can use these components. In this case, it is easy to locate and comprehend a refactored entity to be modified (i.e., *maintainability* increases). Moreover, remodularization provides a support for *extensibility*. The new components can be easily extended to support new capabilities.

However, remodularizations come with a price. They require the comprehension of the system as a whole in order to define the new structure. Program comprehension on legacy systems and architectures in some stage of erosion becomes a challenge. For example, Sarkar et al. [2009] report their experience in the remodularization of a legacy banking system, which had more than 25 MLOC. The project took two years, involving hundreds of developers.

In such a long period, it is difficult to keep enhancements or bug corrections out of the remodularization. In the context of agile development, one may keep adding

functions or correcting bugs for upcoming deadlines and further releases. Murphy-Hill et al. [2009] refer to this kind of activity as *floss refactor*, as opposed to *root-canal refactor* referring to exclusive refactoring changes. Such problem relates to Kent Beck's metaphor of Two Hats [Fowler et al., 1999]: developers perform small refactorings to provide support to a new function, as well as they refactor a new function to make it easier to maintain. During software development, it is common to switch hats and include both refactorings and enhancements.

Other challenge of remodularizations is the scope of their applicability. There are a growing number of tools supporting small refactorings. Some tools—such as the Smalltalk Refactoring Browser [Roberts et al., 1997] and the refactoring menu in Eclipse [Eclipse, 2013]—support small refactorings proposed by Fowler et al. [1999]. Even though these transformations are not totally safe [Soares et al., 2013] and developers do not fully take advantage of refactoring tools [Vakilian et al., 2012], they facilitate the process of applying refactor changes.

However, at the architectural level, there is a need for automation of remodularizations [Fowler et al., 1999; Lippert and Roock, 2006; Bourqun and Keller, 2007; Terra et al., 2012, 2014]. In the remodularization of the banking system reported by Sarkar et al. [2009], some analysis tools were proposed to extract static dependencies such as function calls, but the changes performed in this remodularization were completely manual.

When applying code refactoring, there is a catalog of small scale examples to follow [Fowler et al., 1999; Demeyer et al., 2002]. But in remodularizations, there are few guidelines or patterns to follow when starting to modularize a legacy system. Because of this drawback, remodularization is often applied in an advanced stage of architectural erosion [Anquetil and Laval, 2011].

2.1.1 Modularization Operators

Rama and Patel [2010] formalized six elementary operations that are likely to occur in any remodularization, called *modularization operators*. These operators were recurring in three remodularization cases, including Linux and Mozilla. They also discuss specific situations to which every operator was applied. A short description of each operator is given as follows:

- **Module Decomposition (MD):** the most recurring operator consists in partitioning a big module into smaller ones. Files are distributed over these new modules to create groups of high-related files.

- **Module Union (MU):** the opposite operator of Module Decomposition consists in creating a bigger module from smaller ones. It is often applied when files have similar responsibilities, but they are placed in different modules.
- **File Transferal (FT):** this operator consists in transferring a file to another module. It is similar to a Move Class refactoring in Fowler’s catalog [Fowler et al., 1999]. Module Union and Module Decomposition are then a composition of File Transferal operators: new modules are created and existing files are transferred to these modules.
- **Function Transferal (FuT):** similar to Move Method [Fowler et al., 1999], this operator moves one function from one file to another. It is often applied when a function is misplaced in its current file.
- **Promote Function to API (PF):** this operator defines a “promotion” of a function in its scope. As an example in object-oriented systems, a method is changed from private to public. It is the less common operator in the toolkit. It is applied if a function is generic and it can be required by other classes.
- **Data Structure Transferal (DT):** basically, this operator moves one attribute to another file. It is often performed in conjunction with Function Transferal, sharing similar goals.

Although the authors described algorithms to identify the proposed operators, they do not provide tool support for this purpose. Thereupon, they do not evaluate the benefits (in terms of quality) of applying the proposed operators.

2.2 Software Architecture Reconstruction

It is estimated that up to 60% of software maintenance effort is spent on understanding the code to be modified [Abran et al., 2001]. Particularly in open-source systems, there are few or even no documentation available and updated. All of the knowledge about the system is described in source code. Therefore, understanding the system in order to apply changes becomes a challenge.

There are many approaches that extract a high-level model of software, most of them rely on the reverse engineering of the source code. These approaches are categorized in the literature as Software Architecture Recovery [Ducasse and Pollet, 2009]. By extracting an updated model from the source code, SAR techniques improve the

comprehension of how the code was implemented, the design decisions, and potential architectural problems [van Deursen et al., 2004].

On the other hand, SAR techniques can help architects to propose new architectures based on properties of the source code artifacts. These properties might consider different aspects of relationships: common change [Zimmermann et al., 2003; Beyer and Noack, 2005], high-coupling [Mitchell and Mancoridis, 2006], or implementation of a common concern [Bavota et al., 2013b]. A recent work relies on Information Retrieval techniques to group these artifacts by their textual content [Pollet et al., 2007].

After applying a SAR technique, each artifact is assigned to a property or group. Manual SAR approaches rely on the generation and visualization of properties in source code, rather than propose a new organization of architectural entities. Therefore, after applying these approaches, the expertise of the architect is still necessary to conduct architectural decisions.

2.3 Distribution Map

Distribution Map is a generic visualization approach to visualize source code artifacts according to a given property [Ducasse et al., 2006]. Given two partitions of the same entities set, the *reference* partition denotes a well-known partition, and the *comparison* partition is the one that is discovered by SAR. Typically, the reference partition relates to the namespace organization (or packages, for some object-oriented languages).

In a distribution map, the reference partition is represented as rectangles containing filled squares. Each square represents a source code artifact, generally a file or a class. Finally, the color of a square denotes the property its represented artifact was assigned. Therefore, in our further experiments with software systems, we consider that a class belongs to a package and it is also assigned to a given property. The distribution map shows how the comparison partition (i.e., the class properties) matches the reference partition (i.e., the package distribution).

Figure 2.1 illustrates a simple distribution map generated by the Moose platform [Nierstrasz et al., 2005], from the source code of the Colt project.¹ This map has five properties which relate to the number of lines of code: lighter colors (i.e., yellow, orange, and cyan) denote classes with few lines of code; on the other hand, darker colors (i.e., blue and red) denote classes with more than 75 and 100 lines of code, respectively. From the visualization, we conclude that most of the smaller classes are concentrated

¹<http://acs.lbl.gov/software/colt/>

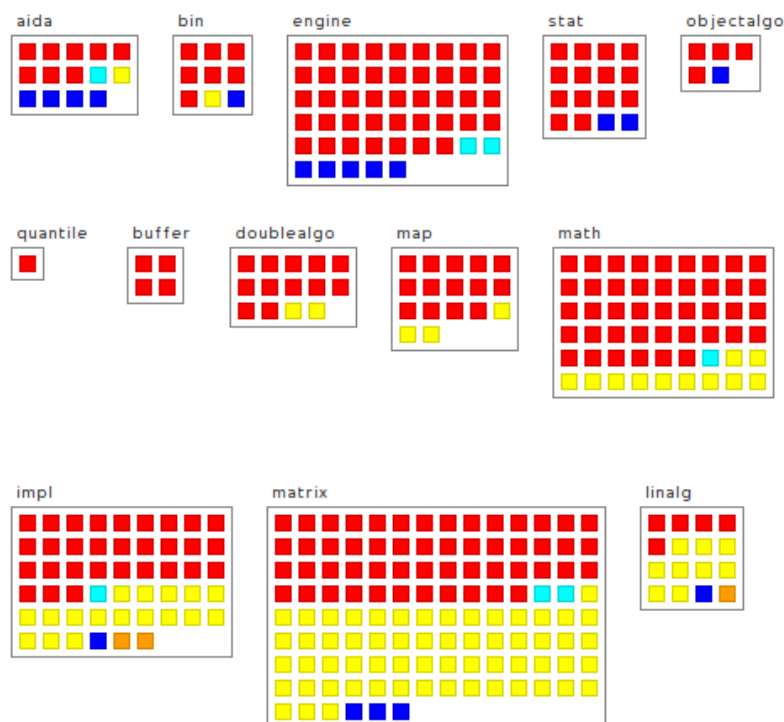


Figure 2.1. Distribution Map of Colt

in the function package. In general, most classes of this project are large, as red is used to fill classes with more than a hundred lines of code.

Ducasse et al. [2006] also proposed two metrics—spread and focus—to calculate the concentration of a property over the reference partition. These metrics are later described in Section 2.7.

2.4 Information Retrieval

Information Retrieval (IR) deals with the representation, storage, organization, and access to information items, usually unstructured documents specified in natural language [Baeza-Yates and Ribeiro-Neto, 2011]. The goal is to provide easy access to information of the user’s interest. Research in information retrieval includes modeling high performances algorithms to process data and language analysis in order to provide relevant query results.

The basic core of any IR system is presented in Figure 2.2. At the preprocessing level (left), the system gathers data in documents. The IR system extracts and filters information in each document, which results in a collection of terms for each document. Terms and documents are then indexed to facilitate the retrieve terms

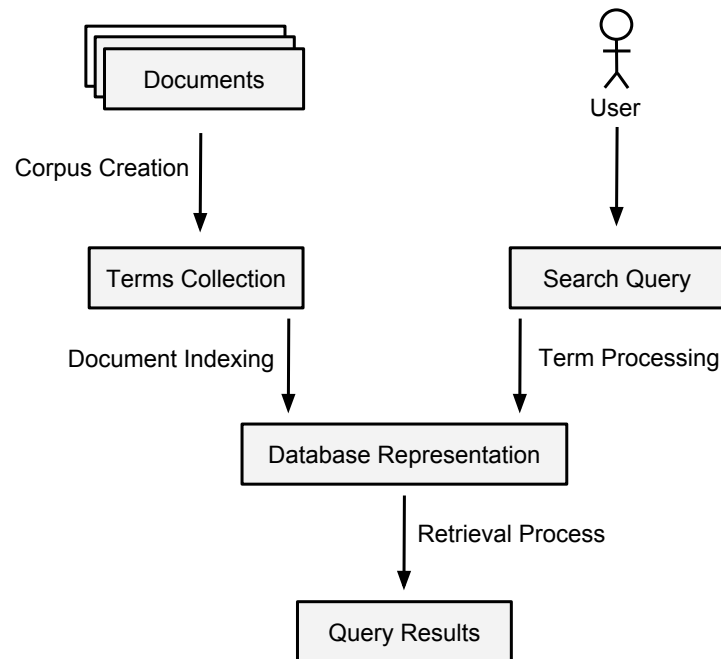


Figure 2.2. Components of an Information Retrieval System

of a given document. Meanwhile, at the user level, the information retrieval begins when a user enters a query into the system. The same filtering process can be done for queries to improve their relevance. The retrieval process consists in gathering the most relevant documents that fits the given query. Each module is detailed as follows:

Crawling is the process of gathering data. In Web applications, this is a complex process that handles with pages, relevance indexes, and hyperlinks. Basically, it involves the selection of relevant *documents* for the database to build a search engine upon it. This process is out of the scope of this dissertation.

In order to assist the understanding of how an IR system works, we present a small example. Suppose we want to extract information from software written in a particular language (for example, Java). In the Web or in private repositories, there are a lot of projects that we need to filter. In the crawling process, a decision is made to determine (i) from which repositories the information will be extracted and (ii) which content of the selected repositories are considered as documents. For example, we should decide to extract information from open source projects in the Qualitas Corpus [Tempero et al., 2010; Terra et al., 2013]. A document will then represent a class of each project.

Definition I — *Document* is a unit of retrieval [Baeza-Yates and Ribeiro-Neto, 2011]. Documents are anything from which one can extract information. They can be Web pages, books, images, or source code files, for example.

Corpus Creation is the process of extracting content from documents as a sequence of characters. The text extraction process relies on the format of each document. For example, source code in a given programming language or XML files have different ways to extract text. The result of this stage is the representation of each document of the database in plain text.

Following the example, the database comprises all classes from each project in the Qualitas Corpus. In order to illustrate how text extraction is performed in each class, we use one class from the Colt project as example: `SparseDoubleMatrix2D`. A summary of this class' source code is shown in Code 2.1.

Code 2.1. Summary of `SparseDoubleMatrix2D`'s source code.

```
public class SparseDoubleMatrix2D extends DoubleMatrix2D {
    protected AbstractIntDoubleMap elements;
    protected int dummy;

    public SparseDoubleMatrix2D(double[][][] values) { ... }
    public SparseDoubleMatrix2D(int rows, int columns) { ... }
    public SparseDoubleMatrix2D(int rows, int columns, ... ) { ... }
    protected SparseDoubleMatrix2D(int rows, int columns, IntDoubleMap ... ) { ... }

    public DoubleMatrix2D assign(double value) { ... }
    public DoubleMatrix2D assign(DoubleFunction function) { ... }
    public DoubleMatrix2D assign(DoubleMatrix2D source) { ... }
    public DoubleMatrix2D assign(DoubleMatrix2D y, DoubleFunction function) { ... }
    public int cardinality() { ... }
    public void ensureCapacity(int minCapacity) { ... }
    public DoubleMatrix2D forEachNonZero(IntIntDoubleFunction function) { ... }
    public double getQuick(int row, int column) { ... }
    protected boolean haveSharedCellsRaw(DoubleMatrix2D other) { ... }
    protected int index(int row, int column) { ... }
    public DoubleMatrix2D like(int rows, int columns) { ... }
    public DoubleMatrix1D like1D(int size) { ... }
    protected DoubleMatrix1D like1D(int size, int offset, int stride) { ... }
    public void setQuick(int row, int column, double value) { ... }
    public void trimToSize() { ... }
    protected DoubleMatrix2D viewSelectionLike(int[] rowOffsets, int[] ... ) { ... }
    public DoubleMatrix1D zMult(DoubleMatrix1D y, DoubleMatrix1D z, ... ) { ... }
    public DoubleMatrix2D zMult(DoubleMatrix2D B, DoubleMatrix2D C, ... ) { ... }
}
```

For corpus creation, we consider the identifiers in this class, including the name of the class and names of its methods and attributes. After creating the corpus, `SparseDoubleMatrix2D` is represented as the text described in Figure 2.3.

```

assign assign assign assign cardinality dummy elements ensureCapacity
forEachNonZero getQuick haveSharedCellsRaw index like like1D like1D
setQuick SparseDoubleMatrix2D SparseDoubleMatrix2D SparseDoubleMatrix2D
SparseDoubleMatrix2D trimToSize viewSelectionLike zMult zMult

```

Figure 2.3. Text extraction of the identifiers of class `SparseDoubleMatrix2D`.

However, the resulting text does not have relevant tokens yet. *Tokenization* is the process of separating entire texts in a set of tokens. Therefore, this process depends on the language the text is described, as each language has its own set of rules for token separation.

Definition II — *Token* is a sequence of characters in particular documents that are grouped together as a useful semantic unit for processing [Manning et al., 2008]. A token might not have meaning until it is properly processed.

For example, in English there are rules to separate tokens with hyphenation and quotation marks, such as `don't`. In programming languages, which does not allow spaces in entity names, identifiers with more than one word are often represented in *camelCase* or *under_score* notation. When applying tokenization in the identifiers from the example class, each word in the camel case notation is separated as depicted in Figure 2.4. For example, the name of the class `SparseDoubleMatrix2D` is separated in the following terms: `sparse`, `double`, `matrix`, and `d`

```

assign assign assign assign capacity cardinality cells d d d d d d double
dummy each elements ensure for get have index like like like like matrix
matrix matrix matrix mult mult non quick quick raw selection set size
shared sparse sparse sparse sparse to trim view z zero

```

Figure 2.4. Tokenization of the text in `SparseDoubleMatrix2D`.

Common words that do not add meaning to the text, called *stopwords*, are also excluded. Some IR systems consider removing small words as well. *Text Normalization* consists in canonicalizing tokens that might be different in form, but have the same meaning. *Stemming* is a common way to normalize tokens, by removing affixes and suffixes of each token. For example, `select`, `selected`, and `selection` have the same

stem `select`. Figure 2.5 shows the result of stemming the text in our example class. For example, the stemming process removes the plural of `cells` and the conjugation of the verb `ensure` (see Figure 2.4 and Figure 2.5 for comparison).

Definition III — *Stem* is the root or main part of a word.

Definition IV — *Term* is a normalized word that is included in the IR system’s vocabulary [Manning et al., 2008].

Definition V — *Vocabulary* is the set of all distinct terms in a document [Baeza-Yates and Ribeiro-Neto, 2011].

```
assign assign assign assign capac cardin cell d d d d d d doubl dummi each
element ensur for get have index like like like like matrix matrix matrix
matrix mult mult non quick quick raw select set size share spars spars
spars spars to trim view z zero
```

Figure 2.5. Stemming of text in `SparseDoubleMatrix2D`.

Term Weighting is the process of assigning a weight for a term in a document. Most of the proposed weighting functions rely on both the term frequency in a document (*term frequency*, or TF) and the frequency of the term in the whole document set (*document frequency*, or DF) [Baeza-Yates and Ribeiro-Neto, 2011]. Luhn [1957] proposed that the more the term occurs in a document, the greater its weight must be in this document. Therefore, the search of a term in the vocabulary will result in the documents that have higher amount of this term.

Definition VI — *Term Frequency* is the sum of occurrences of one term in the document, optionally divided by the amount of terms in the document.

Definition VII — *Document Frequency* is the amount of documents in the database in which a given term occurs at least once [Baeza-Yates and Ribeiro-Neto, 2011].

In the `SparseDoubleMatrix2D` example, the term matrix has its frequency equal to $4/49$, since it occurs four times in this document with 49 terms. Since we are working

with only one document in this example, the document frequency of all terms is equal to one.

Zipf [1932] proposed that the document frequency of terms follows a *power-law* distribution. Thereby, it is common to have terms that occur in most documents of the dataset. These terms are not necessarily relevant to the vocabulary, since they do not distinguish the documents of the dataset [Linstead et al., 2009]. In order to normalize the weight of a term based on its frequency, an *inverse document frequency* is proposed to penalize frequent terms.

Definition VIII — *Inverse Document Frequency*: given the number N of documents in the dataset and the document frequency df_t of a term, IDF is defined as:

$$idf_t = \log \frac{N}{df_t}$$

where \log is the logarithm function at base 2.

Definition IX — *TF-IDF* is the multiplication of the term frequency of the given term in the document and the inverse document frequency of this term in the dataset [Manning et al., 2008].

$$tf-idf_{t,d} = tf_{t,d} \times idf_t$$

Therefore, IDF returns greater values to rare terms in the dataset. On the other hand, the more a term occurs in most of the documents of the dataset, the closer to zero its IDF will be. Finally, TF-IDF returns greater values to terms that most occur in a few set of documents. There are a lot of term weighting functions described in the literature. Most of them are variations of the term and inverse document frequencies [Salton, 1971; Baeza-Yates and Ribeiro-Neto, 2011].

Document Indexing consists in representing the whole database in a search space. The most common way to store this information is in a *Term-Document Matrix*. Each column of this matrix represents a document and each row represents a term in the vocabulary. Therefore, each cell of the matrix represents the weight of the term in the document.

The content of a document can be easily retrieved by a simple extraction of its column in the matrix. Similarly, it is possible to retrieve the collection of documents

that contains a given term, simply by extracting the term’s row in the matrix. This process is called *inverted indexing* because we can reconstruct the text (without order) in the documents using the vocabulary and the term-document matrix [Baeza-Yates and Ribeiro-Neto, 2011].

The resulting matrix of term weights is also called *Vector Space Model*. Each column of the matrix is a vector in a multidimensional space. The dimensionality of this space is equal to the number of terms in the vocabulary. In this case, the direction of a vector is defined by the weights of the terms in its representing document. Weights in a vector space model are usually measured with TF-IDF.

Finally, a search query is programmatically processed and compared to every document of the database; the most similar documents are returned as the query result [Baeza-Yates and Ribeiro-Neto, 2011]. A simple measure of similarity calculates the cosine of the smaller angle between two vectors: a document in the database, by extracting its column in the term-document matrix; and the query as a collection of weighted terms. We can also use a document as a query, in order to return a list of the most similar documents to the given document.

Definition X — *Cosine Similarity* measures the cosine of the angle between two vectors in the same vector space, that is:

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

where $|\vec{v}|$ is the norm of the vector and \bullet is the vector internal product operator. The cosine similarity values vary from 0 to 1.

2.4.1 Latent Semantic Indexing

In the last section, we presented how an Information Retrieval system represents a database as term-document matrices. Although text processing reduces the required space for representation significantly, the resulting matrix might still have hundreds of terms even for few documents. Regarding this issue, Beyer et al. [1999] presented the dimensionality problem: distance measures in vector spaces tend to lose significance when the space has many dimensions (i.e., terms and documents). Moreover, classic algebraic models—such as Vector Space Model—do not consider correlation of terms. Terms are assumed to be mutually independent [Baeza-Yates and Ribeiro-Neto, 2011].

This assumption is not a realistic approximation since terms such as *tv* or *television* are related to the same concept.

Latent Semantic Indexing (LSI) is an IR technique that represents a vector model in a smaller number of terms [Deerwester, 1988]. This dimensionality reduction is provided by a linear algebra technique called Singular Value Decomposition (SVD), which groups terms that occur in the same documents. The term-document matrix is then reduced with minimal loss of information, tackling in this way two linguistic phenomena: different terms with the same meaning (synonymy) and terms with multiple meanings (polysemy) [Deerwester et al., 1990; Baeza-Yates and Ribeiro-Neto, 2011].

Given a term-document matrix, SVD decomposes it in three matrices. We describe the properties of this decomposition as follows:

$$A = U.S.V^T$$

- U is a square matrix, which columns are eigenvectors of $A.A^T$
- S is a diagonal matrix with the eigenvalues (or singular values) of A , in decreasing order
- V is a square matrix, which columns are eigenvectors of $A^T.A$

LSI is not an automated technique because it relies on the number of terms of the reduced matrix, called k . However, Manning et al. [2008] suggest a number in hundreds for a collection of thousands of documents. They also state that the consequences of choosing small values for k do not compromise the similarity between the original and the reduced matrices. Kuhn et al. [2007] propose a method to calculate k as a function of the dimensions of the original matrix. Assuming a given k as input, LSI reduces the original term-document matrix in the following steps:

- Calculate r as the rank of S
- Derive S_k , replacing by zero the $(r - k)$ values from S
- Derive A_k , calculating $A_k = U.S_k.V^T$ and removing the $(r - k)$ last rows of A

The transformation of matrix A is also called the low-rank approximation of A . Any query in the original space must be transformed to the new one by using the decomposed and the reduced matrices. Similarity between documents in the reduced space are calculated in the same way. Basically, a query vector \vec{q} in the vector space defined by matrix A is represented in the new space as follows:

$$\vec{q}_k = S_k^{-1}.U_k^T.\vec{q}$$

2.4.2 Information Retrieval Techniques in Reverse Engineering

Anquetil and Lethbridge [1997] proposed one of the first approaches for software architecture recovery using textual information. They extract *n-grams* as subsets of characters from file names, and they apply clustering to group files with *n-grams* in common. The goal of their work was to evaluate what software engineers consider as subsystems. Other criteria are also considered, such as routine names and comments. The study included one system with 140 files, separated in 10 subsystems by four developers. After that, the clustering result for a given criterion was compared with the developer's configuration. They concluded that the file name criterion is more likely to discover subsystems in a legacy system according to developers.

Maarek et al. [1991] extracted information from attributes in software components to build libraries. They proposed that terms have an "affinity" and build groups of terms with similar meanings. This concept relates to the classification in LSI that is based on co-occurrence. It was also one of the first work to apply hierarchical clustering for classification of software entities. In an experiment with Unix users, the authors compared search query results generated by the proposed approach and by a tool that does not provide term classification. The evaluation provided indications that, according to the user needs, term classification during indexing process provides better precision in search queries without losing recall.

Maletic and Marcus [2000] were among the first to propose the use of LSI and software entities classification with clustering. Their approach was able to retrieve significant groups of files using only textual analysis. Maletic and Marcus [2001] also proposed the comparison between the system's physical structure and the clusters generated from text analysis. In a experiment with a 95 KLOC system with no external documentation, they evaluate how clusters help to improve comprehension and maintenance tasks. They concluded that, although clusters represent specific concepts in the system's domain, they are often spread over multiple files and procedures.

In a second experiment, Marcus et al. [2004] evaluated the precision and recall of user queries over the same system. They concluded that grouping classes or terms outperforms dependency-based approaches and basic text retrieval tools, such as *grep*. However, a combination of concept location approaches with structural and textual information is needed in order to provide better results.

Corazza et al. [2011] analyzed the importance of different lexical information from source code to clustering quality. They propose the use of *zones*, in which terms from different sources of the vocabulary have distinct weights. For example, the vocabulary associated with class names and method names has different weights in the overall

vocabulary. They compare this approach with existing dependency-based clustering approaches with non-extremity and authoritativeness metrics. The approach presented better results with both metrics in 13 systems. However, the weight of each zone is different for each system.

Applications of information retrieval techniques are not only related to architecture recovery. Poshyvanyk et al. [2013] proposed the use of LSI and Formal Concept Analysis to build a search engine applied to source code. The developer describes a search query in natural language and the approach returns a list of source code elements that are more similar to such query. In a recent work, Dit et al. [2013] combined dynamic analysis and web-based IR techniques—such as Page-Rank—to improve the search engine. In an experiment with three real-world systems, the query results had an average effectiveness reaching to 87% above the state of the art.

Gethers et al. [2011] proposed CodeTopics, a tool that relies on information retrieval to support program comprehension. The tool needs a set of high level artifacts (such as requirements or use cases) as input. These artifacts are described in natural language text by developers. Based on the text similarity between high level artifacts and the source code, CodeTopics shows how much the artifacts are actually implemented in the code. In two controlled experiments with students, the authors concluded that this similarity can help developers to improve the quality of source code identifiers, nearing them to the natural language artifacts.

Also regarding program comprehension, Silva et al. [2014] proposed an approach to assess software modularity based on co-change graphs. Basically, the approach defines clusters of classes that are frequently changing together. These clusters are compared to the package distribution in a Distribution Map. The authors also analyzed the correlation between the conceptual cohesion of a co-change cluster, as extracted from the issue description, and its distribution in the packages, as calculated by spread and focus. Although the results are not conclusive to all systems under analysis, they observed that the higher the similarity of the issues in a cluster, the more likely the clusters concentrate inside the packages.

2.5 Semantic Clustering

Semantic Clustering is an architecture recovery technique originally proposed by Kuhn et al. [2005, 2007] that extracts sets of classes in a system. These sets are called *semantic clusters*. Classes are grouped according to the similarity of their vocabularies. Although the term *semantic* is used, the technique originally only

considers the co-occurrence of terms, i.e., a lexical-based relation. Figure 2.6 was extracted from Kuhn et al. [2007] and presents an overview of Semantic Clustering processing. Three main functions are proposed to generate semantic clusters and they are detailed as follows:

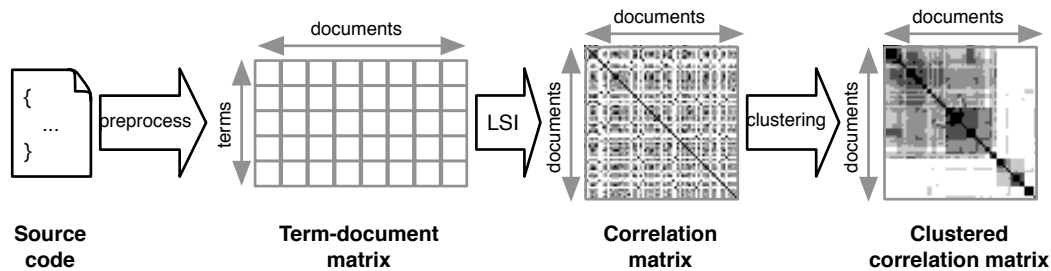


Figure 2.6. Semantic Clustering overview, extracted from Kuhn et al. [2007].

Information Retrieval: As exemplified in Section 2.4, Semantic Clustering consider classes as documents. The vocabulary of a class is extracted from its name, the identifiers of attributes and methods, and the content of comments. Therefore, programming language keywords are discarded. Stopwords removal and stemming are also performed. Terms are weighted with TF-IDF and organized in a term-document matrix. LSI is then performed to reduce the term-document matrix, as well as to group terms that occur together in the same documents.

Clustering: After reducing the term-document matrix with LSI, Semantic Clustering calculates the cosine similarity of each pair of documents in the system and builds a correlation matrix. Using an agglomerative hierarchical approach, the clustering algorithm initially considers each class as a cluster and, at each step, the pair of clusters with the highest average similarity between their elements is merged into a new cluster. Each class is assigned to a single cluster that represents a domain concept.

Visualization: Semantic Clustering uses the semantic clusters and the package distribution to provide a Distribution Map visualization (refer to Section 2.3). The semantic cluster to which a class is assigned denotes the property, i.e., the color of this class in the visualization. An additional information is also provided: a short list of the most relevant terms from the classes in a cluster, called *semantic topics*, which represents the meaning (or intention) of each cluster.

Kuhn et al. [2007] reported experiments with jEdit and JBoss that investigate whether semantic clusters actually express the meaning of the code. Since Semantic Clustering is a visualization technique, the experiments were restricted to nine clusters for each system. The authors concluded that semantic clusters are more likely to discover application, domain-independent concepts, and layers. Moreover, the approach has a tendency to return larger domain-specific clusters. We report in Chapter 3 our adaptation in Semantic Clustering in order to assist modularization analysis.

2.6 Structural Metrics

An extensive number of software metrics has been proposed in the literature. Regarding software architecture, most of the metrics rely on static and structural aspects, such as method calls or use of attributes. In this section, we report studies concerning the evaluation of the applicability of such architectural metrics.

Abreu and Goulão [2001] proposed a procedure to improve modularity using clustering techniques and coupling metrics. A good modularity is the one whose classes are highly-coupled in the proposed modules. They observed that the improved modularization differed from the original one in the number of modules. Therefore, the authors concluded that practitioners do not seem to use only cohesion and coupling as the driving forces when it comes to modularization.

Counsell et al. [2005] formulated a study with 24 developers with different programming experience. These developers evaluated randomly selected classes from real-world applications concerning their cohesion. The goal of the study was to verify whether the perception of cohesive unit agrees with size and coupling metrics. The results suggest that class size does not impact such perception. Moreover, not only coupling is considered, but also a conjunction of number of comments and how the methods of a class implement a given concept in common.

Ó Cinnéide et al. [2012] reported a comparative and quantitative evaluation of software metrics that measure the same aspect of quality: cohesion. They selected five widely used structural cohesion metrics in the literature. After an automated refactoring operation, all metrics are calculated. The goal was to verify whether the value of these metrics evolve together, given the same refactoring. Considering eight systems and over three thousand refactorings, in 38% of the cases at least one metric disagree with each other, i.e., one metric value increased while the other metric value decreased. This fact is an indicative that the structural cohesion metrics measure different and conflicting aspects of the same property.

Regarding real remodularization cases, Anquetil and Laval [2011] used cohesion and coupling metrics over three remodularizations of Eclipse to verify whether the metrics follow the widely recommended quality guideline of high cohesion and low coupling. However, the coupling values increased in most of the packages and the cohesion metric presented a flaw in the experiments. They concluded that either the structural metrics or the cohesion/coupling dogma fails in representing architectural quality.

2.7 Conceptual Metrics

This section presents the conceptual metrics used in this master dissertation. Two of these metrics were proposed by Ducasse et al. [2006] and are applied to Distribution Maps (refer to Section 2.3).

- **Conceptual Cohesion of a Cluster (CCCluster):** This metric is a straightforward extension of the Conceptual Cohesion of a Class (C3) metric, proposed by Marcus and Poshyvanyk [2005]. C3 is calculated as the average cosine similarity of each pair of methods in a given class. Similarly, CCluster is the average cosine similarity of each pair of classes in a *cluster*. Moreover, the internal cohesion of a clustering is the average CCluster of all generated clusters.
- **Conceptual Cohesion of a Package (CCP):** This metric is similar to CCluster metric, but it is defined as the average cosine similarity of each pair of classes in a given *package*.
- **Spread:** Given two multi-sets of classes, the package distribution P , and the collection C of semantic clusters (refer to Section 2.5), we calculate the touch of a cluster in a package as the percentage of classes in this package that were assigned to the cluster:

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

where $c \in C$ and $p \in P$. Basically, spread computes the number of packages in which at least one class is covered by a given cluster, or more formally:

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

where P denotes the set of all packages. We expect a decrease in this metric after a modularization. It is related to change impact, in the sense that a concept must be less scattered to reduce future maintenance work.

- **Focus:** Similarly, the touch of a package in a given cluster is the percentage of classes in the cluster that belong to the package. Therefore, focus measures the percentage of classes one cluster touched in their respective packages.

$$focus(c, P) = \sum_{p \in P} touch(c, p) \times touch(p, c)$$

Focus is a number between 0 and 1 and measures the distance between the clustering and package distributions [Ducasse et al., 2006]. Thus, if the focus is close to one, then the cluster covers the majority of classes in the packages it touches. After a modularization, we expect an increase in focus. The rationale is also related to change analysis: if we have a concept that is concentrated in few packages, then it will be easy to maintain it.

Ducasse et al. [2006] provided a short example of a Distribution Map with five packages, presented in Figure 2.7. Considering the property in red, it covers two packages. Therefore, its spread is equal to two. However, this property does not cover all classes of these two packages. It covers three classes in the package 3, which has five classes, from total 15 classes that were attributed to this property. On the other hand, the property in red also covers 12 classes in the package 4, which has 14 classes, from total 15 classes covered by this property. Thereby, the focus of the property in red is calculated as follows:

$$focus(red, P) = touch(red, P_3) \times touch(P_3, red) + touch(red, P_4) \times touch(P_4, red)$$

$$focus(red, P) = \frac{3}{5} \times \frac{3}{15} + \frac{12}{14} \times \frac{12}{15} = 0.80$$

Wong et al. [2000] also proposes metrics to measure the proximity of a reference partition and the package partition. In their case, the *disparity* metric relates to Spread by penalizing packages which classes are related to different features. Similarly, the *concentration* metric relates to Focus, but it only considers the intersection between

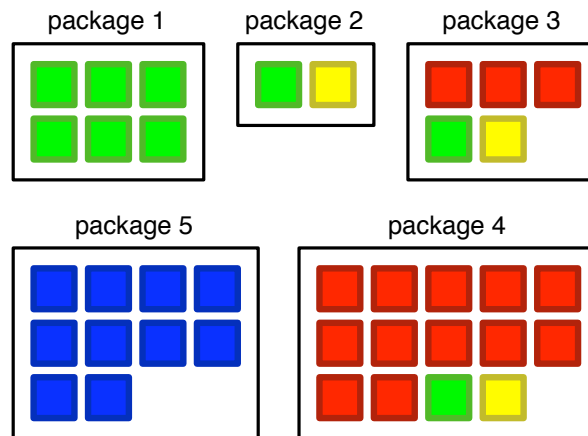


Figure 2.7. Example of a Distribution Map

the classes that were covered by a feature and the classes of a given package. The number of packages to which one property (or feature) spreads or concentrates is not taken into account in both metrics.

Regarding the evaluation of conceptual metrics, Ujhazi et al. [2010] proposed an empirical study with both structural and conceptual coupling metrics to evaluate their accuracy in predicting bugs in Mozilla. The results showed that the proposed conceptual metrics are strongly correlated with the structural metrics, unlike a traditional conceptual metric like C3 [Marcus and Poshyvanyk, 2005]. Also in comparison with C3, the combination of conceptual and structural metrics provided better accuracy and precision than combining C3 with structural metrics.

Recent work in architectural recovery comprise the combination of structural and conceptual metrics. Scanniello et al. [2010] proposed the use of structural links—such as static dependencies—to derive architectural layers. Lexical information is then analyzed to decompose each layer into modules. This decomposition is based on the cosine similarity of the classes’ vocabulary and on the k -means clustering algorithm. In comparison with widely used clustering algorithms, this approach outperformed those algorithms in authoritativeness, i.e., considering the current partition as the authoritative one.

In a simpler approach, Bavota et al. [2010, 2013b] proposed a method to identify Module Decomposition opportunities. Given one module, chains of classes with high coupling are found and a new package is proposed for each chain. Coupling is measured by a combination of a structural and a conceptual metric. They applied the approach in five systems and concluded that the refactoring suggestions are meaningful from the perspective of developers.

2.8 Final Remarks

There is a growing research in program comprehension and software metrics, both with same goals regarding maintenance. However, the challenges in such areas are separated in two main items:

- At the architectural level, there are few findings or lessons learned of how to perform modularization in real software systems. There is still a lack of knowledge of how real modularizations happen and which patterns developers tend to follow in order to improve the architecture.
- Recent work questions the use of traditional structural metrics to measure architectural quality. These metrics either do not agree with each other or do not reflect the improvement of modularizations performed by developers.

We observed in the literature that, concerning both challenges, there is a trend in analyzing the vocabulary of a system to (i) extract and propose a conceptual architecture; or (ii) measure the conceptual quality of the architecture. In this master dissertation, we rely on semantic clustering and conceptual metrics to evaluate real modularizations at the level of modularization operators performed by developers.

Chapter 3

Improvements to Semantic Clustering

This chapter presents our improvements to Semantic Clustering in the following aspects: (i) a set of strategies to filter text that might not add information to the software vocabulary (Section 3.1); (ii) a clustering stop criterion that is not based on a fixed number of clusters (Section 3.2); and (iii) a strategy to generate the same number of clusters of two versions of a software's modularization (Section 3.3). This chapter also presents tool support both to Semantic Clustering and the improvements proposed in this chapter (Section 3.4).

3.1 Text Filtering Strategies

Most IR techniques filter out content from their documents in order to obtain a collection of relevant terms. We presented in Section 2.4 the stopwords filtering, a recurrent strategy to remove terms that do not add meaningful information. When working with source code, we also have to handle programming language keywords. This process is already addressed since we only extract the text in identifiers and comments.

However, the content described in comments contains natural and unstructured language. Originally, Semantic Clustering treats every non-letter character as a word separator. Nevertheless, other information should also be removed. In our experiments with Java systems, we observed that developers often use HTML and annotation tags to insert meta-data. Code 3.1 presents the first lines of source code of class `ControlContribution` extracted from Eclipse. JavaDoc allows the developer to insert metadata in the documentation, e.g., by the tag `@param`, and HTML tags, e.g., by the tag `<code>`. We consider this information as a keyword of the JavaDoc language,

therefore we discard it. Only information about the context of a class (i.e., what does this class do?) is considered.

Code 3.1. Summary of `ControlContribution`'s source code.

```

/**
 * An abstract contribution item implementation for adding an arbitrary
 * SWT control to a tool bar.
 * Note, however, that these items cannot be contributed to menu bars.
 * <p>
 * The <code>createControl</code> framework method must be implemented
 * by concrete subclasses.
 * </p>
 */
public abstract class ControlContribution extends ContributionItem {
    /**
     * Creates a control contribution item with the given id.
     * @param id the contribution item id
     */
    protected ControlContribution(String id) {
        super(id);
    }
    [...]
}

```

For this purpose, we defined a set of regular expressions to filter out meaningless terms, particularly in JavaDoc comments. The regular expressions are described in Table 3.1. For example, we defined filters to treat external content, such as information about the class author (Filter 1) or external URLs (Filter 2). One last filter removes words with less than four characters, which typically denote acronyms, abbreviations, and small (and therefore less significant) words.

The second improvement to text filtering is the definition of a stopword list. In the literature, there is no authoritative stopword list that is applicable to all IR techniques. We propose the use of a stopword list derived from the SMART project.¹ SMART is an information retrieval system developed in the 60s, used as reference by modern search machines [Salton, 1971].

Furthermore, a preliminary study revealed a set of common words specific to Java. For example, classes often overwrite the `hashCode` method. Therefore, this method adds little information about classes. To address this issue, we added 12 words obtained from all public methods in the `Object` class and primitive constants, such as `true` and `null`. These additional stopwords are presented in Table 3.2, resulting in a stopword list with 583 words (i.e., including the SMART list).

¹<http://jmlr.csail.mit.edu/papers/volume5/lewis04a/a11-smart-stop-list/english.stop>

Table 3.1. Regular Expressions for Java Source Code Filtering

1	JavaDocs' author, version and since information <code>@(author version since).*\n</code>
2	URLs <code>(http ftp https)://[\w-]+(\."[\w-]+)+([\w-\.,@?^=%&;;/~+#]*["\w-@?^=%&;;/~+#])?</code>
3	XML's special characters like <code>&amp;</code> <code>&[a-z]*;</code>
4	HTML tags like <code><body></body></code> <code></?[a-z]*></code>
5	Java class declarations <code>java\.[a-z]*\.[a-zA-Z]*</code>
6	Parameter, exception and reference info <code>@(param throws see exception) [^]*</code>
7	'@' annotations <code>@[a-z]*</code>
8	Hexadecimal numbers <code>0x[~]*z</code>
9	Words with less than four characters <code>^[a-zA-Z]{1,3}\$</code>

Table 3.2. Additional Stopwords

object	clone	equals	finalize	hash	code
notify	string	wait	null	true	false

3.2 Semantic Clustering Stop Criterion

By originally proposing Semantic Clustering as a visualization technique, Kuhn et al. [2007] presented a case study in which they propose a fixed number of nine clusters for analysis. Clearly, this number does not scale for large systems. Considering hundreds of classes, the semantic clusters will most likely contain many classes and therefore they will be difficult to analyze. To address this issue, we propose to stop the hierarchical clustering described in Section 2.5 using a *similarity threshold*. In this case, the most similar pair of clusters is merged at each step of the algorithm, until no pair has its similarity greater than a given threshold.

This strategy ensures that different number of clusters can be extracted from different systems. Moreover, it only relies on the similarity between the vocabularies of the classes. For example, if one executes Semantic Clustering in a system whose class vocabularies are loosely similar, a high threshold will make the clustering stop earlier, generating a large number of clusters. On the other hand, lower thresholds underestimate the similarity of classes and therefore tend to generate large clusters, covering most of the system's classes.

Even though we modified the clustering stop criterion, Semantic Clustering still needs an input parameter. We propose to execute the clustering algorithm several times, changing the similarity threshold value. Finally, we recommend to select the threshold that produces highly cohesive clusters and a plausible number of clusters. In Section 4.3, we define the methodology for threshold selection in our experiments.

3.3 Semantic Clusters Generation

As described in Section 2.5, Semantic Clustering uses a distribution map visualization to compare the semantic clusters and the organization of packages of a system. In our approach, we want to compare two architectures, before and after a modularization, under a conceptual point of view. In this case, the number of clusters must be the same, so that we can verify whether conceptual aspects can explain the applied refactorings in the architecture.

A naive solution would be to extract the semantic clusters separately from each version and to compare the generated clusters. However, refactoring activities, such as the creation or deletion of classes, modify the database. Therefore, the vocabulary is not the same, which creates new similarities between the documents. Consequently, the number of clusters before and after a modularization are not likely to be the same. In this case, the semantic clusters from two versions of a modularization might not be comparable.

In order to extract the same conceptual architecture from both versions, we propose an algorithm that supports the generation of semantic clusters after the modularization. Semantic Clustering is applied only to the earlier version of the system. Each class from the newer version (i.e., the version after the modularization) is mapped to a semantic cluster that was previously calculated. The algorithm assigns the class to its most similar cluster, in terms of cosine similarity.

We present the algorithm in Figure 3.1. Basically, it receives as input (i) the semantic clusters from the earlier version of the modularization ($Clusters_{before}$); and (ii) the classes in the newer version (C_{after}). The algorithm generates as output a new list of semantic clusters ($Clusters_{after}$), in which each class in the newer version is mapped to one of the original clusters.

For each new class, we calculate its representing vector in the Vector Space Model (line 6) by extracting its column in the term-document matrix (refer to Section 2.4). Similarly, the algorithm calculates the vector of a cluster as the sum of the vectors of its classes (line 7). Then, the algorithm tests the cosine similarity of a class and each

Require: $Clusters_{before}, C_{after}$
Ensure: $Clusters_{after}$

- 1: $Clusters_{after} = \emptyset$
- 2: **for** $c \in C_{after}$ **do**
- 3: $bestSimilarity = -\infty$
- 4: $bestCluster = -1$
- 5: **for** $cluster \in Clusters_{before}$ **do**
- 6: $\vec{v}_{class} = classAsVector(c)$
- 7: $\vec{v}_{cluster} = clusterAsVector(cluster)$
- 8: $s = cosineSimilarity(\vec{v}_{cluster}, \vec{v}_{class})$
- 9: **if** $s > bestSimilarity$ **then**
- 10: $bestSimilarity = s$
- 11: $bestCluster = cluster.index$
- 12: **end if**
- 13: **end for**
- 14: $assign(Clusters_{after}(bestCluster), c)$
- 15: **end for**

Figure 3.1. Semantic Clusters Generation Algorithm

cluster (line 8). Finally, the new class is assigned to the cluster that returns the highest similarity in the computed tests (line 14). With this strategy, it is possible that no class from the newer version will be assigned to one particular cluster. However, in our experiments, this situation did not occur.

3.3.1 Visualization Support

The semantic clusters comparison algorithm also provides the visualization in distribution maps, as presented in Figure 3.2. In this visualization, there are two distribution maps, for the earlier and newer versions. The location of the classes and packages is fixed. If a package (or class) was created after the modularization, then it is displayed as blank in the first distribution map. Similarly, if a package or class was removed after the modularization, it is displayed as blank in the second distribution map.

Figure 3.2 displays three packages in the modularization of JHotDraw. The Figure does not show all packages and clusters of each version. The first column represents the packages before the modularization and the second column represents the same packages after the modularization. Eleven classes were created in `org.jhotdraw.app` package, and twelve classes were removed. The `org.jhotdraw.draw.tool` package was created with twenty classes. And finally, `org.apache.batik.ext.awt` was completely removed in this modularization.

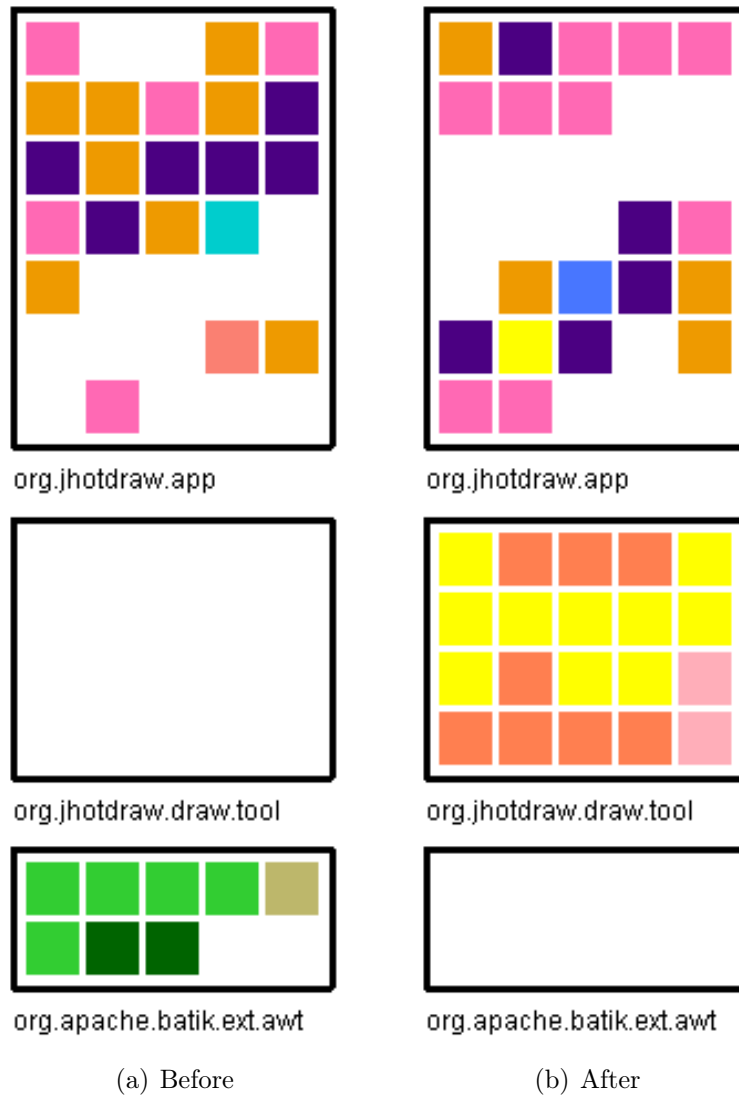


Figure 3.2. Distribution Maps in the remodularization of JHotDraw

At this point, we only consider creation, transferral, and removal operations. The concentration of semantic clusters after a remodularization is measured by the conceptual metrics, previously described in Section 2.7.

3.4 Tool Support

3.4.1 Topic Viewer

We implemented a prototype tool, called TopicViewer [Santos et al., 2013], that supports our improvements to Semantic Clustering. TopicViewer is a desktop application implemented in Java that provides:

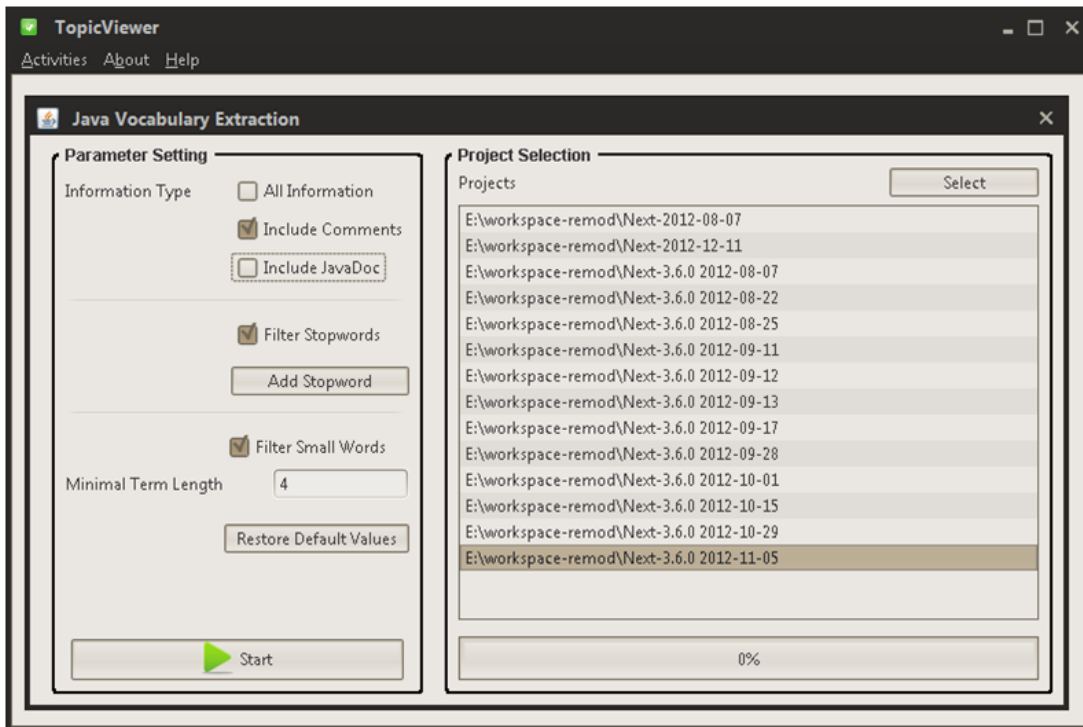
- *Text Extraction* from Java source code;
- *Semantic Clustering* operations, i.e., LSI indexing and hierarchical clustering with similarity threshold;
- *Distribution Map displaying*, including the comparison view, in which the user can navigate through packages and classes;
- *Conceptual Metrics measurement* for the metrics we use in this master dissertation.

Figure 3.3 shows the user interface for TopicViewer. In Figure 3.3a, the user provides as input the folder in which the source code is stored (right panel). Finally, Figure 3.3b shows the Distribution Map Viewer, in which the user interacts with the map structure and inspects the conceptual metrics results.

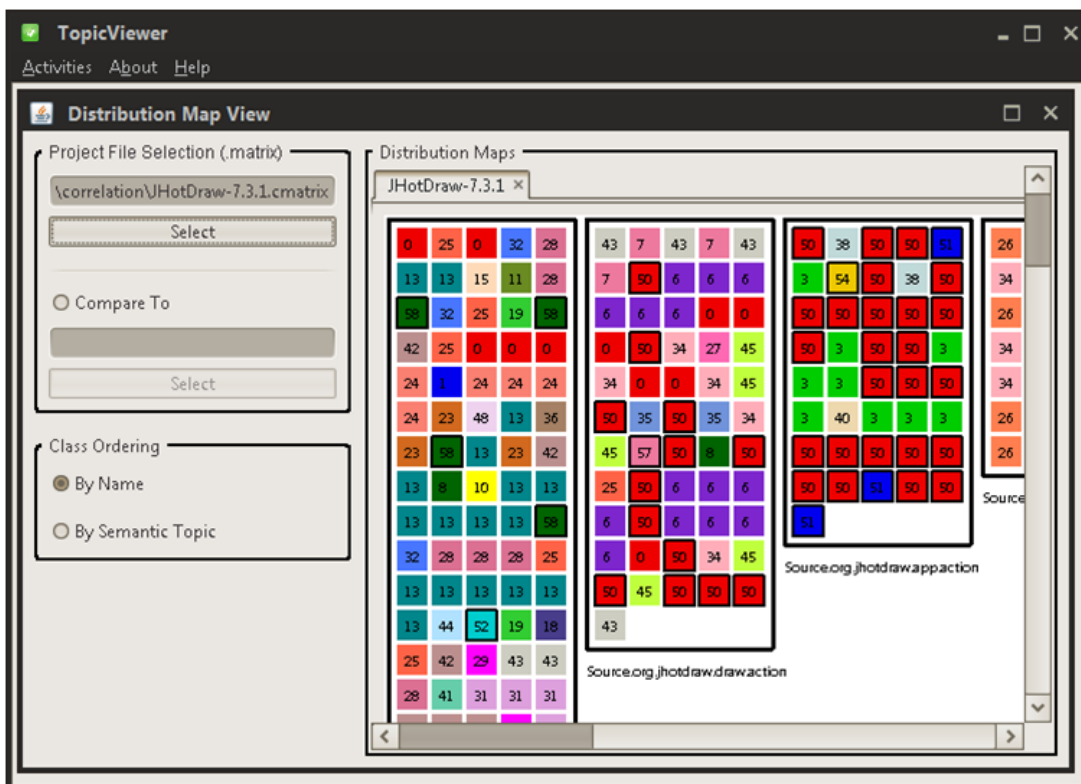
The current TopicViewer implementation follows a Model-View-Controller architectural pattern, as presented in Figure 3.4. The *Extraction* view (as displayed in Figure 3.3a) is responsible for the extracting, filtering, and indexing terms from source code. For this purpose, the tool reuses basic IR functions from VocabularyTools [Santos et al., 2012], developed with the Software Practices Laboratory of Federal University of Campina Grande (UFCG). The functions supported by VocabularyTools include text extraction, filtering, and LSI operations. The *Semantic Clustering* view provides a configurable environment for the user to set the similarity thresholds. The tool can also estimate the best threshold, following the methodology that will be described later in Section 4.3. Finally, the *Distribution Map* view (as displayed in Figure 3.3b) provides the visualization, interaction, and metrics measurement of the generated clusters in a Distribution Map.

TopicViewer’s architecture provides an extensible interface for representing term-document matrices. Although we focused on text extraction from source code, other textual documents such as bug reports can also be processed. In order to integrate new text extractors to the tool, one only needs to implement how these sources are translated into rows and columns of the matrix. For example, TopicViewer was recently used in another work to support the visualization of co-change clusters [Silva et al., 2014]. Moreover, the operations in the term-document matrix are executed in file, which scales for large collections of documents. The latest version of TopicViewer is available at GoogleCode.²

²<http://code.google.com/p/topic-viewer>.



(a) Java Vocabulary Extraction



(b) Distribution Map View

Figure 3.3. TopicViewer's User Interface

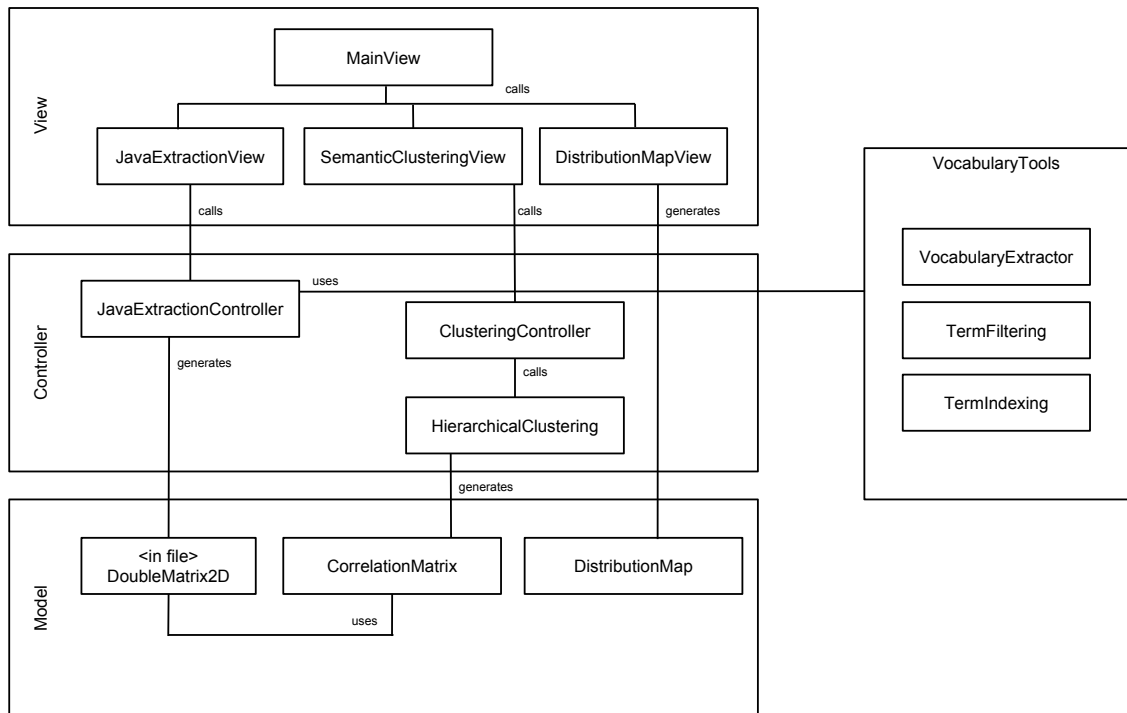


Figure 3.4. TopicViewer’s Architecture

3.4.2 Hapax 2.0

In comparison with the state of the practice, TopicViewer is similar to Hapax, a plug-in for the Moose platform [Nierstrasz et al., 2005]. Hapax supports Semantic Clustering as originally proposed by Kuhn et al. [2007]. However, since its first release, Hapax was discontinued due to maintenance in Moose and Pharo—a Smalltalk dialect. During a two-months internship at RMoD/INRIA Lille, we upgraded Hapax to the latest configuration of Moose and Pharo (5.0 and 3.0, respectively).

We also improved Hapax with text filtering and similarity threshold strategies, both described in this chapter, in order to generate feasible clusters for analysis. Figure 3.5 presents the visualization of semantic clusters in a distribution map with Hapax. The latest version of Hapax is available at SmalltalkHub.³

³<http://smalltalkhub.com/#!/GustavoSantos/Hapax>

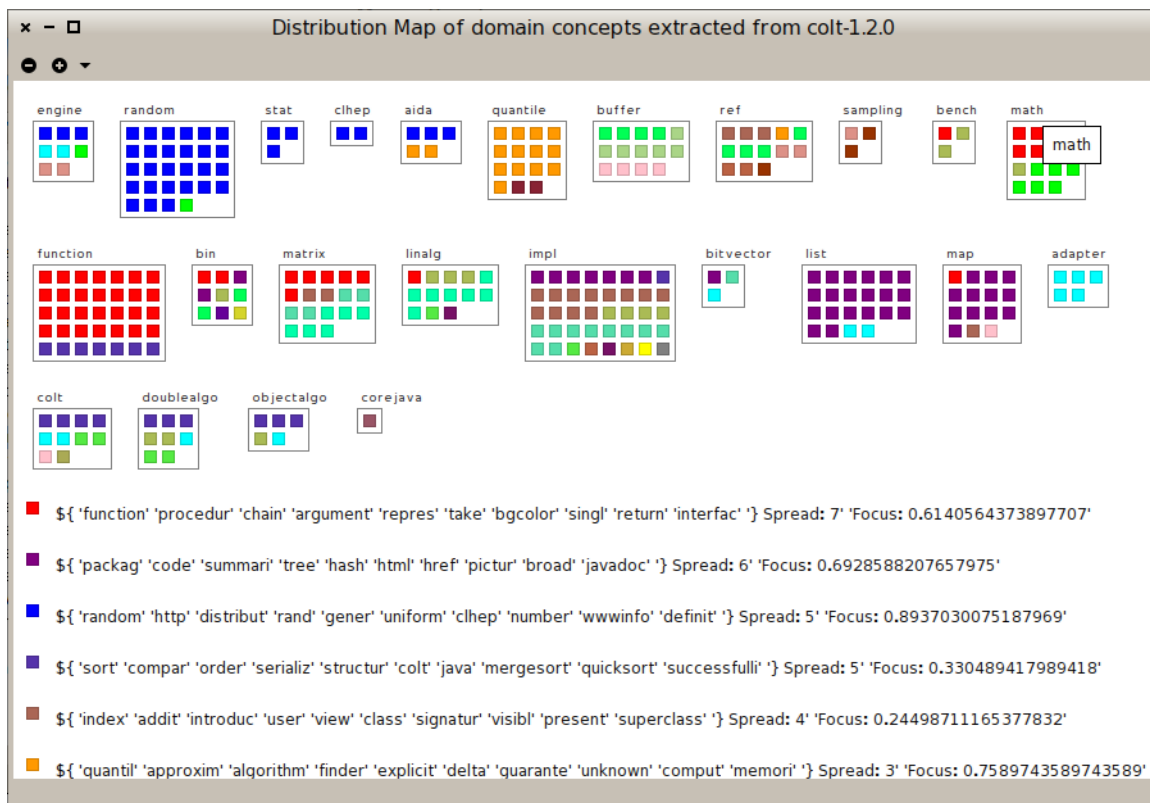


Figure 3.5. Distribution Map visualization in Hapax 2.0

3.5 Final Remarks

In this chapter, we proposed a collection of improvements to Semantic Clustering, a technique originally proposed to support program comprehension [Kuhn et al., 2005, 2007]. Our improvements included a set of text filtering strategies, the clustering stop criterion based on the similarity of the vocabulary, and a strategy to support the analysis of two versions of a system, under a conceptual point of view.

The contributions of our work also include tool support with prototypes in two environments: (i) TopicViewer, a desktop tool which supports our improvements to Semantic Clustering in order to assist modularization analysis; and (ii) Hapax 2.0, a Moose plug-in which supports the original Semantic Clustering approach, currently available as a plug-in for the latest version of Moose.

Chapter 4

Evaluation

In this chapter, we report on a case study in which we analyze the remodularizations of real software systems using the proposed improvements to Semantic Clustering. We follow a comparative and quantitative methodology proposed by Anquetil and Laval [2011]. According to this methodology, we assume that real remodularizations improve the internal quality of a software system, i.e., the newer version is “*better*” than the previous one. We stand in this assumption because the analyzed remodularizations were performed by the system’s developers for a considerable amount of time. More important, the resulting architecture was adopted for the newer versions. This fact means that the new architecture is at least “as good as” the previous one.

Section 4.1 presents the remodularizations under analysis in our study and Section 4.2 describes the preparation and cleaning of our dataset. We separate our evaluation of Semantic Clustering and conceptual metrics in two parts. The first part describes the methodology we followed to configure the parameters of Semantic Clustering (Section 4.3). Finally, the second part presents the study on the remodularizations and conceptual metrics (Section 4.4). We also report on threats to validity of our approach in Section 4.5 and final remarks on the evaluation in Section 4.6.

4.1 Target Systems

Anquetil and Laval [2011] evaluated two remodularizations of Eclipse regarding structural metrics for cohesion and coupling. In our study, we added remodularizations of three Java-based systems by recommendation. After that, our study totalizes six remodularization and four case systems:

- **Eclipse** went through a substantial remodularization to integrate the OSGi technology. Existing features of Eclipse were separated into new components during two remodularizations: from version 2.0.1 to 2.1 and from 2.1 to 3.0. The Eclipse case consists in a major and global remodularization. For example, the component responsible for providing programming interfaces, named `ui`, was separated into five components in version 3.0.
- **JHotDraw** also had two remodularizations: from version 7.3.1 to 7.4.1 and from 7.4.1 to 7.5.1. The first remodularization is global, which comprised the separation of two packages into 16 new packages. On the other hand, the last one is local because it impacted only two packages.
- **NextFramework** is a web-based development framework.¹ The system's remodularization is very similar to Eclipse since it also happened to move to the OSGi technology. The remodularization was globally applied to the system's architecture.
- **Vivo** is an open-source researcher networking and collaborative platform.² The remodularization we considered was restricted to the subsystem Vitro, from version 1.4.1 to 1.5. The main changes in this remodularization comprised the specification of web items and the separation of two packages, named `util.pageDataGetter` and `search.solr`. Thus, this remodularization is similar to JHotDraw's one: a local restructuring changing a small number of packages.

Table 4.1 provides descriptive statistics of these systems and their versions. For each one, we calculated the vocabulary considering identifiers, i.e., class, attributes, and method names, and also comments and documentation (V_{all}). Moreover, we calculated the average number of terms per class ($V_{\text{all}}/\text{NOC}$).

4.2 Methodology

This section details the methodology we followed in the preparation of each remodularization of our dataset. Other analysis—such as the comparison of semantic clusters—is specific to each case study.

¹<http://www.nextframework.org>.

²<http://vivoweb.org>.

Table 4.1. Vocabulary Data (NOC= number of classes; NOP= number of packages)

System	Release Date	KLOC	NOP	NOC	V _{all}	V _{all} /NOC
Eclipse 2.0.1	29/08/2002	420	104	2,331	3,414	1.46
Eclipse 2.1	27/03/2003	494	110	2,620	3,771	1.44
Eclipse 3.0	25/06/2004	599	142	3,138	3,741	1.19
JHotDraw 7.3.1	18/10/2009	126	46	715	1,878	2.63
JHotDraw 7.4.1	17/01/2010	125	62	715	1,807	2.53
JHotDraw 7.5.1	01/08/2010	134	64	748	1,856	2.48
Next 12-08-07	07/08/2012	56	52	536	1,449	2.70
Next 12-08-22	11/12/2012	67	73	607	1,487	2.45
Vivo 1.4.1	07/02/2012	142	91	899	1,902	2.12
Vivo 1.5	12/07/2012	147	95	940	1,920	2.04

Isolating the remodularization: In order to attend the definition of remodularization proposed in Chapter 2, we first isolated the modifications restricted to architecture from enhancements or new features. Therefore, we manually inspected each package that was created after the remodularization. New packages that introduce new features were discarded from our analysis. Test classes were discarded as well. The data presented in Table 4.1 refer to the systems after this filtering process.

For example, in Eclipse and Next, we observed the creation of plug-ins to integrate these systems to OSGi technology. Particularly in the Next, external libraries were also transformed into new components. Naturally, the integration of a new technology can impact in the architecture for it to adapt to this technology. However, this integration was well organized in packages, which we discarded from our analysis. In the OSGi case, one entire package was responsible for the configuration of this technology, i.e., the definition of components and the connection between them. In this case, the changes in the architecture of Eclipse and Next for them to use OSGi included the remodularization of these systems. In other words, the remodularization in these systems was a means for them to attend to OSGi specifications. On the other hand, in the Next's libraries case, all packages of each library were transferred as new packages of Next. The dependencies to these libraries stayed the same.

However, we could not apply such filtering at the level of classes, e.g., a class that includes a new feature and it is placed in an existing package. This kind of inspection requires the expertise of a developer to manually pinpoint which features are implemented in the source code.

Semantic Clustering: After preparing the dataset, we executed Semantic Clustering to measure the conceptual metrics. In order to configure Semantic Clustering, a distinct similarity threshold was selected for each system. This threshold selection is later described in Section 4.3. After that, the semantic clusters generation algorithm was performed to retrieve the same number of clusters from two versions of a system (refer to Section 3.3). As originally proposed by [Kuhn et al., 2007], we consider the reference partition—to which the semantic clusters will be compared in a Distribution Map—as the distribution of packages. Although this distribution might not be considered as architecture, we emphasize that we focused on finding real remodularization cases. Moreover, in open-source systems, the distribution of packages is the closest and the most current to structural design one can find. However, it is in our interest to provide the definition of architectural components as input to Semantic Clustering.

Computing conceptual metrics: For each remodularization, we calculated the conceptual metrics in the versions before and after the remodularization. The metrics have already been described in Section 2.7 and our analysis is further reported in Section 4.4.

4.3 First Study: Semantic Clustering Setup

In Chapter 3 we described our improvements to semantic clustering in order to apply this technique on remodularization analysis. However, it is crucial to our approach to select the similarity thresholds used by the clustering algorithm. We also discussed the importance of this threshold to correctly estimate the similarity of the classes' vocabulary. Particularly, we want to avoid the occurrence of few large clusters or a large number of small clusters.

To address this issue, we propose to execute the clustering algorithm several times, changing only the similarity thresholds in increments of 0.05. More specifically, we executed the algorithm with five thresholds: 0.55, 0.60, 0.65, 0.70, and 0.75. After that, the quality of the resulting clusters should be evaluated to balance two measures:

- **Internal Cohesion:** The goal is to extract highly cohesive clusters. For this purpose, in Section 2.7 we describe an internal cohesion metric that can be applied to evaluate the clusters generated by Semantic Clustering. The clusters that maximize this metric value are candidates for the analysis.
- **Number of Clusters:** On the other hand, cohesion is not the only criterion to evaluate cluster quality, otherwise we can foster the generation of many small

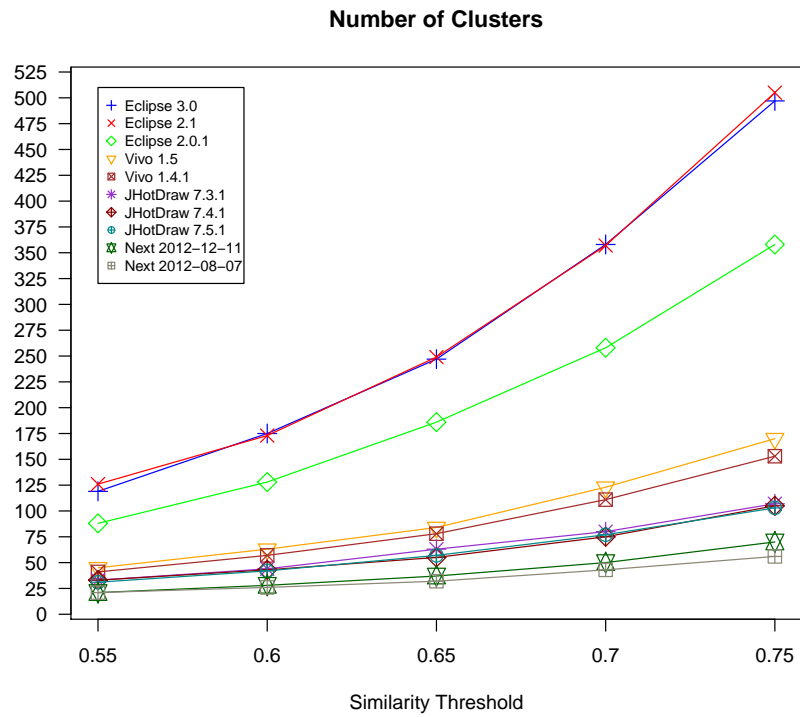
clusters. It is not reasonable to require software maintainers to analyze hundreds of clusters, even when they are highly cohesive. For this reason, we must choose a similarity threshold that minimizes cohesion, but at the same time produces a reasonable number of clusters.

We applied this methodology to each system of our dataset. The first criterion—internal cohesion—is automated and basically selects the threshold which generated the most cohesive clusters in average. The second criterion (i.e., consider the number of generated clusters) was manually followed, based on the experience of the authors and also on the average number of classes per cluster. Figure 4.1a shows the number of clusters for each tested threshold. Then, Figure 4.1b shows the average internal cohesion of the resulting clusters, calculated according to the CCCluster metric. Our findings in this study are reported as follows:

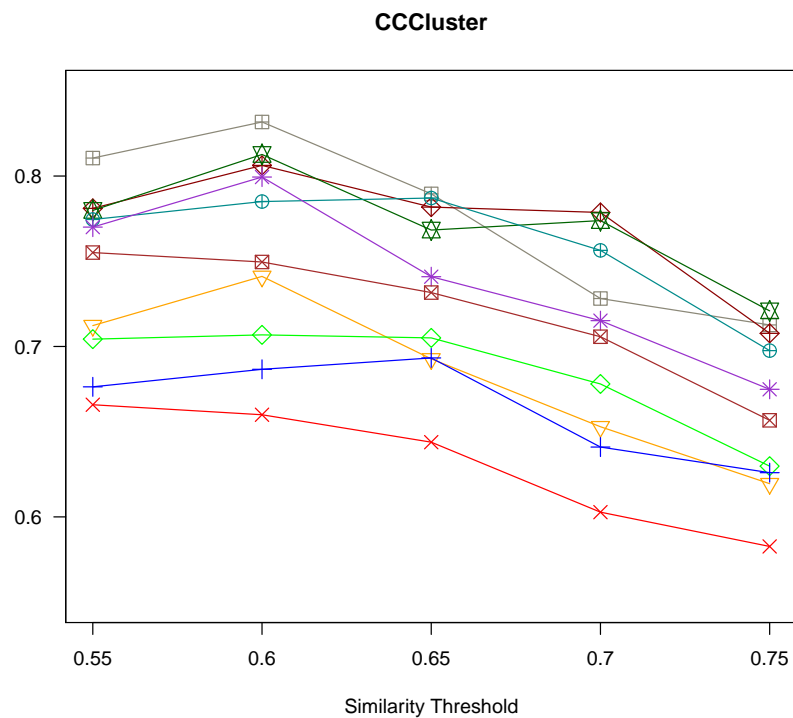
High Thresholds, More Clusters: As expected, the higher the similarity threshold, the larger the number of clusters. For example, from threshold 0.70 to 0.75, we obtained a difference from 80 to 107 clusters with JHotDraw-7.3.1. We also observed that this difference correlates with the vocabulary size and the number of classes. For example, also from threshold 0.70 to 0.75, we obtained a difference from 358 to 497 clusters with Eclipse 3.0, and from 357 to 505 clusters with Eclipse 2.1. These systems are exactly the ones with more classes and larger vocabulary in our dataset (refer to Table 4.1). Although they differ considerably in the number of classes, they are similar regarding the size of vocabulary and also in the number of generated clusters.

The diversity of the vocabulary (i.e., the number of distinct terms) tends to increase the dissimilarity of the classes because there are more terms that distinguish one class to another. Therefore, a high threshold overestimates the similarity of the vocabulary. In this case, when the classes are loosely similar, a slight adjustment in the similarity threshold may abruptly change the number of clusters.

Different Versions, Different Number of Clusters: Even with almost the same number of classes from version 7.3.1 to 7.4.1 (see Table 4.1), the number of clusters in these two versions of JHotDraw are not the same. For example, with threshold 0.65, the number of clusters decreased from 63 to 55. This difference is an indicative that a significant change happened in the vocabulary of JHotDraw, at the point of modifying the similarity between its classes. This fact emphasizes the importance of our algorithm to generate semantic clusters to the version after the modularization (as described in Section 3.3).



(a) Number of Clusters



(b) Internal Cohesion

Figure 4.1. Number of clusters (a) and Internal Cohesion (b) of clusters for similarity thresholds ranging from 0.55 to 0.75 (discrete intervals of 0.05)

Different Versions, Different Degree of Similarity: Similar to the number of clusters, the average cohesion of clusters also differs considerably with different versions of a system. For example, with threshold 0.65, we obtained an increase in cohesion from 0.74 to 0.78 in JHotDraw 7.3.1 and 7.4.1, respectively. This difference means that the classes are more similar inside their clusters. This fact also explains why there are less clusters in JHotDraw 7.4.1.

As a result of this first study, Table 4.2 presents the thresholds we recommend for each system in our dataset. In eight out of ten systems, the thresholds are lower or equal to 0.60. Although we cannot generalize these results, we claim that thresholds respecting this range are at least the first values that should be considered when applying Semantic Clustering in other systems.

Regarding the number of generated clusters, we only applied this criterion to one case. In Eclipse 3.0, a similarity threshold of 0.65 resulted in the best cluster cohesion, but also in 247 clusters. We then chose the second best threshold, 0.60, which resulted in 175 clusters. The first configuration generated many small and highly cohesive clusters, which is justified by the 247 clusters that were generated. This amount is more than the double the number of cluster in version 2.1 (109), and an average of 12 classes per cluster. In the second configuration, the average cluster cohesion is equal to the previous version (0.68), but resulted in larger clusters: an average of 18 classes per cluster. In Section 4.4.2, we analyze the classes of a set of semantic clusters for manual inspection of code. In this case, we decided that around 20 classes per cluster is a reasonable number for qualitative analysis.

Table 4.2. Threshold Selection (Clu= # clusters)

System	Threshold	Clu	CCCluster	NOC / Clu
Eclipse 2.0.1	0.60	132	0.71	18
Eclipse 2.1	0.55	109	0.68	25
Eclipse 3.0	0.60	175	0.68	18
JHotDraw 7.3.1	0.60	44	0.79	17
JHotDraw 7.4.1	0.60	43	0.80	17
JHotDraw 7.5.1	0.65	57	0.78	14
Next 12-08-07	0.60	27	0.80	20
Next 12-12-11	0.60	22	0.83	28
Vivo 1.4.1	0.55	34	0.75	27
Vivo 1.5	0.65	75	0.76	13

4.4 Second Study: Remodularization Analysis

The goal of this study is to verify whether conceptual aspects, as expressed by semantic clusters and conceptual metrics, actually express an increasing in quality after real remodularizations. Thereby, we analyze the remodularizations by (i) identifying major changes in conceptual metric values and (ii) explaining these changes under the perspective of typical modularization operators (refer to Section 2.1.1). More specifically, we intend to provide insights for the following research questions:

- RQ #1: *What is the impact of remodularizations in the clusters generated by Semantic Clustering?* Basically, we aim to compare the semantic clusters before and after remodularizations, using focus and spread. Section 4.4.1 provides our answers and insights regarding this question.
- RQ #2: *What are the modularization operators that have more impact in the clusters generated by Semantic Clustering?* In the first question, we identify the major impacts in spread and focus. Therefore, in this question we intend to classify which modularization operators were responsible by such impact. Section 4.4.2 provides our answers and insights regarding this question.
- RQ #3: *What is the impact of module decomposition in terms of conceptual cohesion?* We intend to establish correlations between the most recurrent remodularization operator—Module Decomposition—and conceptual cohesion. Section 4.4.3 provides our answers and insights regarding this question.

4.4.1 Impact of Remodularizations on Semantic Clusters

After setting the parameters up of Semantic Clustering for our dataset in Section 4.3, we performed the semantic clusters comparison algorithm for each remodularization. We restricted this analysis to global remodularizations (refer to Section 4.1) because they provided the major changes in terms of clusters. After executing the algorithm, we were able to compare the semantic clusters for the versions before and after remodularization. Therefore, for each cluster we calculated its spread and focus in both versions.

Figures 4.2 and Figure 4.3 present the values of spread and focus, before and after each global remodularization. Each point in the scatterplots represents a semantic cluster. The horizontal axis represents the metric values *before* the remodularization and the vertical axis represent the values after the remodularization. A point above the diagonal line means that the metric value for its representing cluster increased after

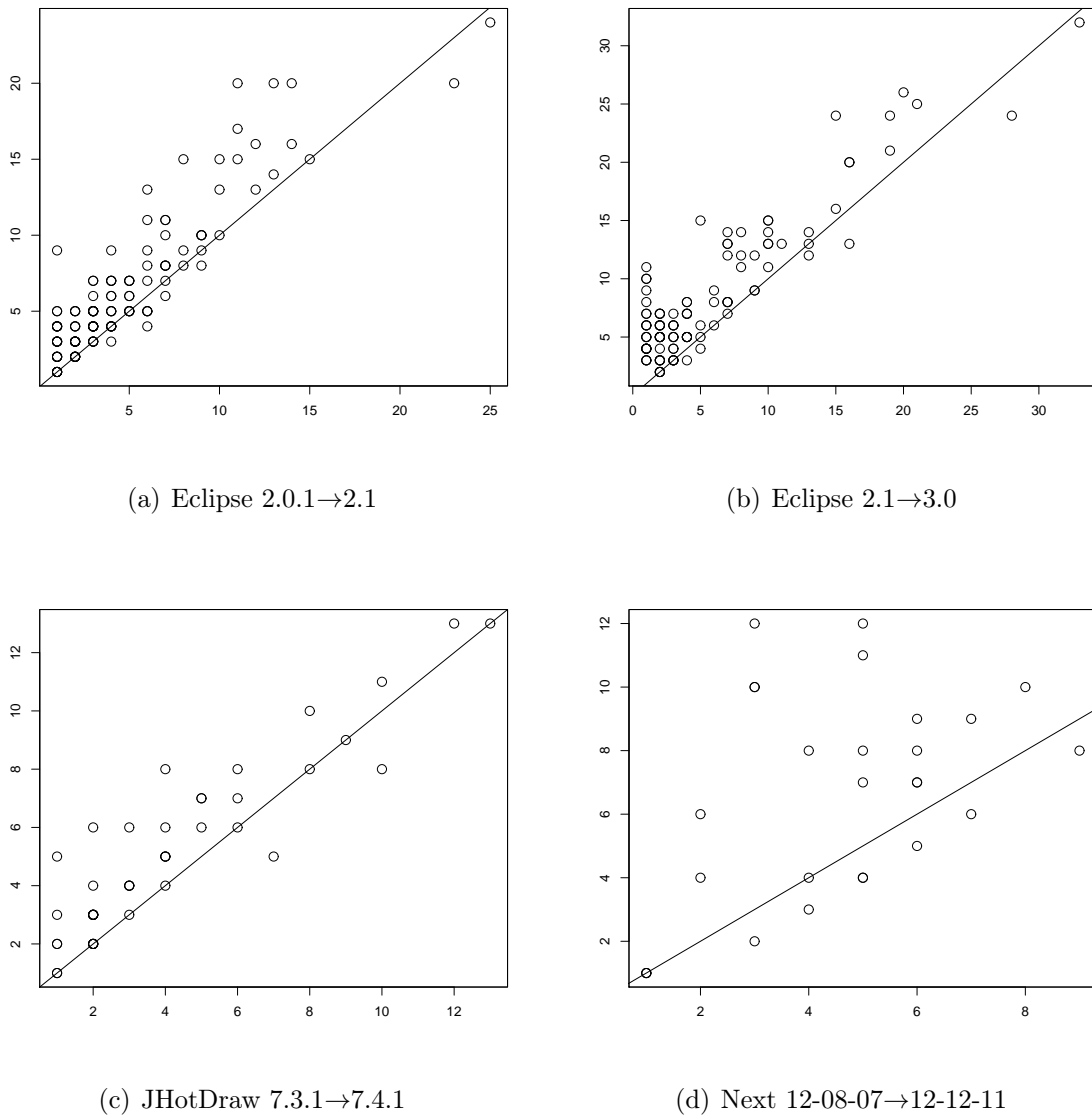


Figure 4.2. Spread results (the spread of the concepts *above* the diagonal increased after the remodularization)

the remodularization. On the other hand, a point below the diagonal line means that the metric value decreased.

For most clusters, spread increased after the remodularization, as most of the points are above the diagonal line. This fact seems counter-intuitive, since we expected that the concepts would be more organized (i.e., covering less packages) after the remodularization. However, this increasing is natural given our data setting. The number of clusters is the same before and after the remodularization, and new packages were created in all cases (see Table 4.1). Therefore, we naturally observe that existing

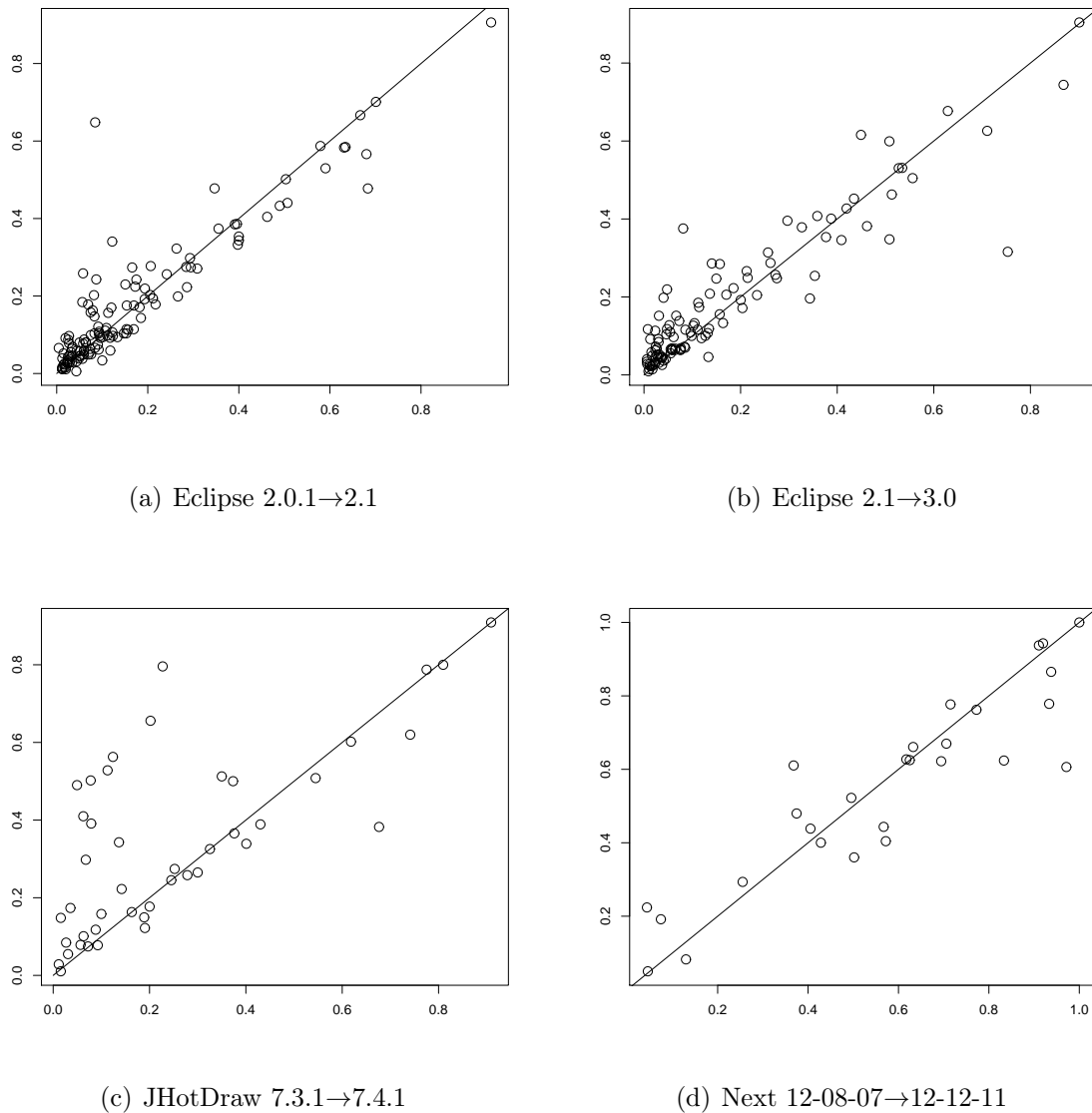


Figure 4.3. Focus results (the focus of the concepts *above* the diagonal increased after the remodularization)

clusters will appear in more packages after the remodularization. We do not consider that an increase in Spread express also an increase in quality. However, Spread was able to express that, in the package creation, the concepts are not completely transferred from one package to another. They split into more packages.

Concerning focus, we observe in Figure 4.3 that the distributions are not clear, since we have a considerable number of clusters both above and below the diagonal line in all remodularizations under analysis. Therefore, the results for focus are not conclusive in this experiment setting.

We applied a nonparametric Wilcoxon test to compare the remodularizations, according to both metrics. This test calculates whether two numerical populations of the same size are nonidentical. In this case, we compare the semantic clusters before and after a remodularization, according to spread and focus separately. After the Wilcoxon test, if the p -value is lower than 0.05, then the clusters are statistically nonidentical and we can claim that the remodularization actually changed the conceptual structure of the system, in terms of the metric under analysis.

Table 4.3 presents the Wilcoxon test results. Values in bold denote that the calculated p -value is lower than the threshold and therefore a statistically significant value. The results show that the increasing of spread is statistically significant in the entire dataset. On the other hand, confirming our previous observation, we cannot assure the same regarding focus. The Wilcoxon results only provide statistical support to the last remodularization of Eclipse and the first remodularization of JHotDraw, in which there was an increase in focus in both cases.

Table 4.3. Wilcoxon test results for spread and focus of clusters

Metric	Remodularization	Mean Increase	p-value
Spread	Eclipse 2.0.1→2.1	1.61	< 0.001
	Eclipse 2.1→3.0	2.82	< 0.001
	JHotDraw 7.3.1→7.4.1	0.98	< 0.001
	Next 12-08-07→12-12-11	2.04	0.001
Focus	Eclipse 2.0.1→2.1	0.01	0.408
	Eclipse 2.1→3.0	0.02	< 0.001
	JHotDraw 7.3.1→7.4.1	0.09	0.008
	Next 12-08-07→12-12-11	-0.02	0.518

Summary: Remodularizations tend to consistently increase the spread of the existing semantic clusters among the new package structure when considering the same number of clusters. Regarding focus, there is no clear tendency, and it is possible to have clusters both with an increase or with a decrease in focus. Therefore, focus is not an appropriate quality metric to support remodularization analysis.

4.4.2 Remodularization Operators with the Highest Impact in Semantic Clusters

In order to address this research question, we focused on the modularization operators proposed by Rama and Patel [2010] (see Section 2.1.1). For each remodularization, we

collected the three clusters with the highest increase and decrease in spread and focus, separately. We aim to identify which modularization operators are able to explain the change in spread or focus. Thus, we carefully analyzed the source code of the classes in each selected cluster and we mapped source code changes to modularization operators, when possible.

To reduce the amount of manual analysis, we decided to inspect one remodularization of each system. For this reason, we removed two remodularizations: (i) Eclipse 2.0.1→2.1 because it presented the highest p -value regarding the global results, as showed in Table 4.3; and (ii) JHotDraw 7.4.1→7.5.1 because it consisted in just local and minor remodularizations.

Table 4.4 summarizes the results of our analysis of 47 clusters (2 metrics vs 6 clusters vs 4 systems). A description of each cluster is detailed in [Santos et al., 2014]. We only found two clusters with bottom results for spread in JHotDraw, i.e., the spread of the other clusters remained constant or increased after the remodularization. For each cluster, we present a brief description of the concepts behind the cluster and the remodularization operator responsible for the change under analysis, when it was possible to identify such operator. For example, we were not able to explain the bottom results for focus in JHotDraw in terms of modularization operators.

We also applied a Pearson chi-squared test to analyze the association of (a) the occurrence of a modularization operator and (b) an increase or decrease in one metric, considering the 47 cases in Table 4.4. After the test, if the p -value is lower than 0.05, these two descriptive variables a and b are dependent. We only applied this test for Module Decomposition because it was the most recurrent operator in the experiment. Other operators have few occurrences, which it is not recommended for a the chi-squared test. Our main findings when investigating this research question are described as follows:

- Module Decomposition (MD) was the operator responsible for most distinguished changes in spread and focus, covering 24 out of 47 clusters we selected for analysis. The operator was responsible for an increase in spread in 9 out of 12 clusters. Regarding focus, the operator explains the observed increments in focus in 10 out of 12 clusters we manually inspected. The chi-squared test showed that the occurrence of module decomposition (a) has statistic association with the Top-3 increasing of spread (b_1) and focus (b_2), with p -values 0.022 and 0.001, respectively.

Table 4.4. Modularization operators responsible for the Top-3 and the Bottom-3 changes in spread and focus (MD= module decomposition; MU= module Union; FT= file transferal; DT= data structure transferal; RN= rename; PF= promote function; MR= module removal; FR= file removal)

Metric	Ranking	System	Cluster id	Operators
Spread	Top 3	Eclipse 2.1→3.0	105	MD
			14	MD, FT, DT
			61	
		JHotDraw 7.3.1→7.4.1	27	MD
			34	MD
			40	MD
	Next 12-08-07→12-11	26	MD	
		25		
		12	MD	
	Vivo 1.4.1→1.5	18	MD	
		16	MD	
		29	MD	
	Bottom 3	Eclipse 2.1→3.0	101	FT, DT
			54	MD
			38	
JHotDraw 7.3.1→7.4.1		17		
		38	MR	
		1	MD, FT	
Next 12-08-07→12-11		3	FR	
		8		
		33	RN, FR	
Vivo 1.4.1→1.5	10	MU		
	5	MU, MD		
Focus	Top 3	Eclipse 2.1→3.0	51	MU, MD
			33	MD, FT
			12	MD, PF
		JHotDraw 7.3.1→7.4.1	34	MD
			12	MD
			16	MD
	Next 12-08-07→12-11	14	MD	
		9		
		22	MD	
	Vivo 1.4.1→1.5	18	MD	
		9		
		22	MD	
	Bottom 3	Eclipse 2.1→3.0	39	FT, DT
			94	MD
			6	
JHotDraw 7.3.1→7.4.1		20		
		23		
		32		
Next 12-08-07→12-11		4	RN	
		21		
		16	RN, MU	
Vivo 1.4.1→1.5	32	FR		
	13	RN		
	28			

- The transferal of files (FT) and data structures (DT) was identified in five cases. In two of them, there was a decrease in spread. This fact confirms the motivation of these operators: moving one entity that is misplaced in the architecture to a more similar module.
- Rename (RN) was performed in Next and Vivo to correct typos, e.g., *Sumary* and *PropStmt*, to *Summary* and *PropertyStatement*, respectively. There was a decrease in focus in three out of four rename operations because new similarities were built with other classes in which the terms are correctly spelled.
- Module Union (MU) occurred along with an increase (with Module Decomposition) and decrease in Focus (with Rename). Promote Function (PF) was applied only once, with an increase in focus also with Module Decomposition.
- We also identified other operations: file (FR) and module (MR) removal in four cases. They were necessary to provide a better interface for color gradients in JHotDraw and code loaders in Next. In these specific cases, there was a decrease in spread.

Summary: Module decomposition is commonly the operator behind the top increasing in spread and focus. This fact means that the semantic clusters cover more packages, but they are also more concentrated inside these packages. For other operators, we cannot draw statistic relationship between the operator and an improvement or decline of a metric.

4.4.3 Impacts of Module Decomposition in Conceptual Cohesion

In the last research question, we aim to investigate the impact of the module decomposition operator in the conceptual cohesion of packages. We focus on this operator because it was the most recurrent operator regarding the highest increase of spread and focus. When performing module decomposition, an original package P is decomposed in new packages P_1, P_2, \dots, P_n . Some of the classes in P are distributed among the new packages, but part of them remain in the restructured package P' . We analyzed all 21 distinct module decompositions in Table 4.4 (one decomposition can impact more than one cluster). We analyzed them under two aspects:

- **Paired Comparisons:** We compared the original package with its restructured version. Figure 4.4a reports a scatterplot with the values for CCP. Each point in the scatterplot represents one module decomposition. The horizontal axis shows the cohesion of the original packages (P) and the vertical axis shows the cohesion of the restructured packages (P'). A point above the diagonal means that the cohesion of the restructured packages is greater than the cohesion of the original ones, i.e., the cohesion improved after the remodularization.
- **New Packages:** We compared the average cohesion of the new packages with the original package in Figure 4.4b. Similar to Figure 4.4a, each point in the scatterplot represents one module decomposition and the horizontal axis shows the cohesion of the original packages (P). The vertical axis shows the average cohesion of the new packages (P_1, P_2, \dots, P_n) in each module decomposition. A point above the diagonal means that the average cohesion of the new packages is greater than the cohesion of the original package, i.e., the module decomposition improved the cohesion, comparing to the former modularization.

For most decompositions, the cohesion of the restructured package improved. We applied Wilcoxon test to compare the two populations of packages—original and restructured—and concluded that there is a significant increase in cohesion of the restructured packages, with p -value equals to 0.002 (<0.05). As well as for the global results, the first remodularization of JHotDraw (i.e., from version 7.3.1 to 7.4.1) presented the best results in both comparisons. For example, the package `draw` had the best improve in conceptual cohesion (+0.111); moreover its ten new subpackages have an average cohesion of 0.818, which is also a very good measure. A similar case occurred in Eclipse, in which a decomposition in the `ui.ide` plugin involved moving classes to the packages in the `ui.workbench` plugin. As a result, these packages increased their cohesion in 0.016 and 0.048, respectively.

Regarding the new packages, the cohesion improved in most of the cases (15 out of 21), but we observed more decreases in cohesion. According to Wilcoxon test, the two populations of packages—original and average cohesion of the new ones—are statistically different and, therefore, the increasing in conceptual cohesion is significant with p -value equals to 0.002 (>0.05). Vivo was one of the systems with significant decrease in conceptual cohesion after module decompositions (-0.084). For example, the new `dataGetter` package is less cohesive than its original package, `pageDataGetter`. This fact is explained by the use of a new interface in the restructured package, which has more utility classes and therefore a larger vocabulary. Thus, the similarity between the classes changed to use this new interface and the rest of the package has decreased.

The same action happened with the package `context` in Next, which resulted in the highest decrease we observed in cohesion considering the new packages (-0.219). In both cases, CCP was able to express that not every module decomposition conceptually improves the packages. In particular cases, when developers generalize interfaces, they directly impact the cohesion in the packages. The cohesion metric might relate to the size of vocabulary and the number of classes in a package, i.e., the more classes in a package, the less cohesive it will be. However, such study is not the scope of this dissertation.

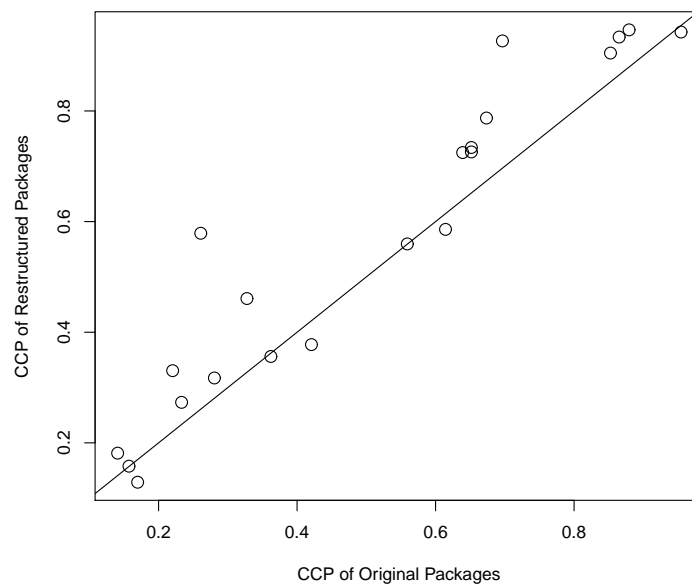
Summary: After module decompositions, the new packages have better conceptual cohesion than the original ones. CCP is an adequate metric to express a quality improvement, but some care must be taken in interpretation (e.g., Next and Vivo cases).

4.5 Threats to Validity

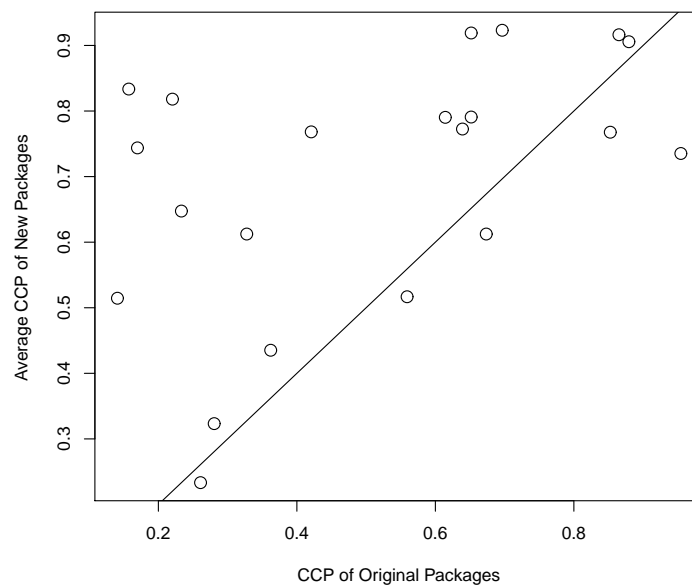
We organized the threats to the validity of our evaluation in four categories, as proposed by Wohlin et al. [2000]:

Construct Validity: The first assumption of our study is that the architecture resulted by the modularizations we observed is always better than the previous one. This assumption is very strong and also very hard to formally prove. The analysis of an expert would be extremely time-consuming, subjective, and non replicable. However, we followed the reasoning discussed by Anquetil and Laval [2011], in which we consider the effort of the developers in an widely-used, open source code, and a considerable amount of time to perform the modularization (refer to Table 4.1). Therefore, it is hard to consider that, after months of structural refactoring, the new architecture is worse than before. In fact, in all cases, the resulted architecture was followed by the following versions.

When collecting the code for our dataset, we do not consider the correction of bugs between two versions. We argue that bug corrections usually impact small portions of code—such as the body of a method, whose vocabulary is not extracted—and, therefore, are not likely to significantly modify the vocabulary. We focused on removing new features because we wanted the two versions of a modularization to have the same number of features (or concepts). In this case, it is also noteworthy that this process was made manually, which is prone to errors and incompleteness.



(a) Conceptual Cohesion of Restructured Packages



(b) Average Conceptual Cohesion of New Packages

Figure 4.4. Conceptual Cohesion results of restructured packages (a) and new packages (b). The cohesion of the packages *above* the diagonal line increased after the remodularization

Also regarding the data collection, we rely on the basic definition of architecture

as the distribution of packages. Although we recognize that this distribution might not correspond to a well-defined architectural design, such discovery task would require the evaluation of experts of each system, which we do not have access. The package distribution is then the closest to a structured design in systems with few (or no) external documentation.

Finally, as any information retrieval technique, Semantic Clustering relies on vocabulary quality. Thus, we assume that terms are well described in the system's identifiers and comments. Naming conventions other than *camelCase* and *under_score* will certainly compromise the vocabulary. The representativeness of the terms in the vocabulary can also impact on our results. We identified in previous studies the existence of methods like “kaboom” or “nothing”. In Section 4.4.2, we also report the refactoring of classes to remove typos in identifiers. In both cases, the vocabulary is polluted with terms that do not often appear and also do not have proper meaning. In this work, we selected systems with a considerable developer community. Therefore, we consider these issues an exception in the vocabulary.

Internal Validity: In our evaluation, we only consider modularization the ones responsible for the conceptual metrics results. Other actions should be considered. For example, the cohesion of a package can increase if the developer improves the vocabulary of its classes. Section 4.4.2 describes the case in which a rename refactoring decreased the focus when correcting typos. In the absence of a metric that quantifies the quality of the vocabulary, we discard this measure from the variables of our study. However, it is in our interest to analyze the meaning of terms in the vocabulary and whether they are also considered when performing modularizations.

External Validity: In this dissertation, we focused on modularization cases that were performed by their own developers. Although we report the difficulty in finding modularization cases, the fact that we collected widely-used systems distinguishes the contribution of this study.

Conclusion Validity: Regarding the analysis of the results, we do not consider the creation of semantic clusters from one version to another. In order to analyze the impact in Spread and Focus, we generated the same number of clusters for the version before and after the modularization. In this case, we needed to automate the process of mapping the same clusters in both versions. A manual mapping would likely be biased and error prone.

Regarding the metrics we used, no study has been done to analyze the reliability

of these metrics. We already discussed that there is few evaluation on the relevance of conceptual metrics applied to software maintenance. Therefore, we chose the metrics that were more easy to explain their variations.

Finally, we manually identified the modularization operators that were performed in the clusters with highest increase and decrease of Spread and Focus. This task is biased and prone to errors. We recognize that these operations are not easy to automate. They also require an advanced knowledge of each system, being a time spending task to perform manually. However, we did make an effort to follow a methodology, and we also tried to isolate the remodularizations from bug corrections and enhancements, in order to gather convincing results from our study with conceptual metrics.

4.6 Final Remarks

In this chapter, we reported our case study with real remodularizations and conceptual metrics. As already discussed in related work, module decomposition is the most common operator when performing remodularizations. Regarding this operator only, it increases the focus of the semantic clusters and the conceptual cohesion of the new packages. This fact reveals that developers organize the system's classes in packages according to a common intent, or concept.

Chapter 5

Conclusion

5.1 Contributions

After decades of research on quality measurement in software engineering, a vast number of quality metrics have been proposed in the literature [Anquetil and Laval, 2011; Ó Cinnéide et al., 2012]. However, there are still few evaluation of these metrics [Briand et al., 1998; Abreu and Goulão, 2001; Ó Cinnéide et al., 2012]. More specifically, we lack an evaluation on their ability to express the architectural improvement after a modularization effort, under the developers' point of view. Clearly, translating a mental model of what constitutes architectural quality to a developer is complex and subjective. Yet, metrics that approximate this perception are important to assist on time-consuming maintenance tasks.

In this work, we selected a subset of recently proposed metrics, named conceptual metrics [Marcus and Poshyvanyk, 2005; Ujhazi et al., 2010]. We also proposed and implemented adaptations to Semantic Clustering in order to support the comparison of clusters, before and after a modularization. We implemented a visualization support to our approach with: (i) TopicViewer, which provides the comparison of two versions of a system according to text analysis, including the measurement of conceptual metrics; and (ii) Hapax 2.0, which supports Semantic Clustering as originally proposed.

After implementing our methodology, we measured spread and focus over a dataset of six modularizations of four real-world software systems. We observed an increase in spread, which means that the concepts cover more packages after the considered modularizations. As stated in previous work [Bavota et al., 2010; Rama and Patel, 2010], module decomposition is the most common modularization operator. The observed increase in spread only confirmed the consequences of using this operator in the distribution of concepts.

However, considering only module decomposition, it was also responsible to the top increases in focus. In other words, the concepts are more spread, but they are also more concentrated in the new packages. We also observed that conceptual cohesion increased in the restructured and in the new packages after the modularizations. This fact reveals that developers organize the system's classes in packages according to a common intent or concept, resulting in more cohesive packages.

The conceptual metrics we used were able to describe an improvement in most of the cases in which module decomposition was performed. Other modularization operators were also analyzed, but they did not have enough occurrence for a proper statistical analysis. Our results reinforce previous work in suggesting module decomposition operations based on conceptual metrics [Bavota et al., 2010, 2013b]. They also reinforce the usefulness of conceptual metrics on expressing the developers' intent when performing modularizations.

5.2 Publications

This master thesis generated the following publications:

- Gustavo Santos, Marco Tulio Valente, and Nicolas Anquetil. Remodularization Analysis using Semantic Clustering. In 1st CSMR-WCRE Software Evolution Week, pages 1-10, 2014.
- Gustavo Santos, Katyusco de Farias Santos, Marco Tulio Valente, Dalton Serey, and Nicolas Anquetil. Topicviewer: Evaluating Remodularizations using Semantic Clustering. In IV Congresso Brasileiro de Software: Teoria e Prática (Sessão de Ferramentas), pages 1-6, 2013.

5.3 Future Work

When investigating the modularization operators [Rama and Patel, 2010], we had the limitation of performing the inspection manually. As future work, we recommend a tool to automatically inspect two versions of a system and to emulate this transformation as basic modularization operators. This approach would present more findings about the modularization operators and the consequences of applying them according to conceptual quality.

The findings of our study also encourage us to extend it at the point of designing a tool to recommend modularization operations based on conceptual metrics. Similar

to previous work on module decomposition, the developer would select which parts of the system are candidates to maintenance. The tool would then calculate a set of conceptual metrics in order to identify remodularization opportunities that might improve the underlying system's concepts. Based on the results, the tool should recommend a set of modularization operators to restore the conceptual quality. For example, the tool may recommend to move a class to a package with a more similar vocabulary.

Bibliography

- Abebe, S. L., Haiduc, S., Marcus, A., Tonella, P., and Antoniol, G. (2009). Analyzing the evolution of the source code vocabulary. In *13th European Conference on Software Maintenance and Reengineering*, pages 189–198.
- Abran, A., Bourque, P., Dupuis, R., and Moore, J. W. (2001). *Guide to the Software Engineering Body of Knowledge*. IEEE Press.
- Abreu, F. B. and Goulão, M. (2001). Coupling and cohesion as modularization drivers: Are we being over-persuaded? In *5th European Conference on Software Maintenance and Reengineering*, pages 47–57.
- Anquetil, N. and Laval, J. (2011). Legacy software restructuring: Analyzing a concrete case. In *15th European Conference on Software Maintenance and Reengineering*, pages 279–286.
- Anquetil, N. and Lethbridge, T. (1997). File clustering using naming conventions for legacy systems. In *1997 Conference of the Centre for Advanced Studies on Collaborative research*, pages 1–12.
- Baeza-Yates, R. A. and Ribeiro-Neto, B. A. (2011). *Modern Information Retrieval, Second edition*. Pearson Education.
- Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., and De Lucia, A. (2013a). An empirical study on the developers’ perception of software coupling. In *35th International Conference on Software Engineering*, pages 692–701.
- Bavota, G., Lucia, A. D., Marcus, A., and Oliveto, R. (2010). Software re-modularization based on structural and semantic metrics. In *17th Working Conference on Reverse Engineering*, pages 195–204.
- Bavota, G., Lucia, A. D., Marcus, A., and Oliveto, R. (2013b). Using structural and semantic measures to improve software modularization. *Empirical Software Engineering*, 18(5):901–932.

- Bayer, J. (2004). *View based software documentation*. PhD thesis, University of Kaiserslautern.
- Beyer, D. and Noack, A. (2005). Clustering software artifacts based on frequent common changes. In *13th International Workshop on Program Comprehension*, pages 259–268.
- Beyer, K. S., Goldstein, J., Ramakrishnan, R., and Shaft, U. (1999). When is “nearest neighbor” meaningful? In *7th International Conference on Database Theory*, pages 217–235.
- Bird, J. (2012). What Refactoring is, and what it isn’t - according to Kent Beck and Martin Fowler. <http://agile.dzone.com/articles/what-refactoring-and-what-it-0/>.
- Bourqun, F. and Keller, R. K. (2007). High-impact refactoring based on architecture violations. In *11th European Conference on Software Maintenance and Reengineering*, pages 149–158.
- Briand, L. C., Daly, J. W., and Wüst, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, pages 65–117.
- Corazza, A., Martino, S. D., Maggio, V., and Scanniello, G. (2011). Investigating the use of lexical information for software system clustering. In *15th European Conference on Software Maintenance and Reengineering*, pages 35–44.
- Counsell, S., Swift, S., Tucker, A., and Mendes, E. (2005). Object-oriented cohesion as a surrogate of software comprehension: an empirical study. In *5th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 161–169.
- da Silva, B. C., Sant’Anna, C., Chavez, C., and Garcia, A. (2012). Concern-based cohesion: Unveiling a hidden dimension of cohesion measurement. In *20th International Conference on Program Comprehension*, pages 103–112.
- Deerwester, S. (1988). Improving Information Retrieval with Latent Semantic Indexing. In *51st American Society for Information Science Annual Meeting*.
- Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of The American Society for Information Science*, 41(6):391–407.

- Demeyer, S., Ducasse, S., and Nierstrasz, O. (2002). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, San Francisco, CA, USA.
- Dit, B., Revelle, M., and Poshyvanyk, D. (2013). Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309.
- Ducasse, S., Gîrba, T., and Kuhn, A. (2006). Distribution map. In *22nd International Conference on Software Maintenance*, pages 203–212.
- Ducasse, S. and Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, pages 573–591.
- Eclipse (2013). The eclipse foundation website. <http://www.eclipse.org/>.
- Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
- Garlan, D. (2000). Software architecture: A roadmap. In *Conference on The Future of Software Engineering*, pages 91–101.
- Garlan, D. and Perry, D. E. (1995). Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274.
- Gethers, M., Savage, T., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. (2011). Codetopics: which topic am I coding now? In *33rd International Conference on Software Engineering*, pages 1034–1036.
- Knodel, J. and Popescu, D. (2007). A comparison of static architecture compliance checking approaches. In *6th Working Conference on Software Architecture*, pages 1–12.
- Kuhn, A., Ducasse, S., and Gîrba, T. (2005). Enriching reverse engineering with semantic clustering. In *12th Working Conference on Reverse Engineering*, pages 133–142.
- Kuhn, A., Ducasse, S., and Gîrba, T. (2007). Semantic clustering: Identifying topics in source code. *Information and Software Technology*, pages 230–243.
- Lehman, M. M. (1996). Laws of software evolution revisited. In *5th European Workshop on Software Process Technology*, volume 1149, pages 108–124.

- Linstead, E., Hughes, L., Lopes, C., and Baldi, P. (2009). Exploring java software vocabulary: A search and mining perspective. In *1st Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 29–32.
- Lippert, M. and Rook, S. (2006). *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley.
- Luhn, H. P. (1957). A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of Research and Development*, 1(4):309–317.
- Maarek, Y., Berry, D., and Kaiser, G. (1991). An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813.
- Maffort, C., Valente, M. T., Anquetil, N., Hora, A., and Bigonha, M. (2013). Heuristics for discovering architectural violations. In *20th Working Conference on Reverse Engineering*, pages 222–231.
- Maletic, J. I. and Marcus, A. (2000). Using latent semantic analysis to identify similarities in source code to support program understanding. In *12th International Conference on Tools with Artificial Intelligence*, pages 46–53.
- Maletic, J. I. and Marcus, A. (2001). Supporting program comprehension using semantic and structural information. In *23rd International Conference on Software Engineering*, pages 103–112.
- Mancoridis, S., Mitchell, B. S., Chen, Y., and Gansner, E. R. (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures. In *15th International Conference on Software Maintenance*, pages 50–59.
- Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
- Marcus, A. and Poshyvanyk, D. (2005). The conceptual cohesion of classes. In *21st International Conference on Software Maintenance*, pages 133–142.
- Marcus, A., Sergeev, A., Rajlich, V., and Maletic, J. I. (2004). An information retrieval approach to concept location in source code. In *11th Working Conference on Reverse Engineering*, pages 214–223.
- Mitchell, B. and Mancoridis, S. (2001). Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *17th International Conference on Software Maintenance*, pages 744–753.

- Mitchell, B. S. and Mancoridis, S. (2006). On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208.
- Murphy, G. C., Notkin, D., and Sullivan, K. (1995). Software reflexion models: Bridging the gap between source and high-level models. In *3rd Symposium on Foundations of Software Engineering*, pages 18–28.
- Murphy-Hill, E., Parnin, C., and Black, A. P. (2009). How we refactor, and how we know it. In *31st International Conference on Software Engineering*, pages 287–297.
- Nierstrasz, O., Ducasse, S., and Gırba, T. (2005). The story of Moose: an agile reengineering environment. In *10th European Software Engineering Conference*, pages 1–10.
- Ó Cinnéide, M., Tratt, L., Harman, M., Counsell, S., and Hemati Moghadam, I. (2012). Experimental assessment of software metrics using automated refactoring. In *6th International Symposium on Empirical Software Engineering and Measurement*, pages 49–58.
- Passos, L., Terra, R., Diniz, R., Valente, M. T., and Mendonca, N. C. (2010). Static architecture conformance checking – an illustrative overview. *IEEE Software*, 27(5):82–89.
- Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52.
- Pollet, D., Ducasse, S., Poyet, L., Alloui, I., Cimpan, S., and Verjus, H. (2007). Towards a process-oriented software architecture reconstruction taxonomy. In *11th European Conference on Software Maintenance and Reengineering*, pages 137–148.
- Poshyvanyk, D., Gethers, M., and Marcus, A. (2013). Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology*, pages 1–34.
- Rama, G. M. and Patel, N. (2010). Software modularization operators. In *26th International Conference on Software Maintenance*, pages 1–10.
- Roberts, D., Brant, J., and Johnson, R. (1997). A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263.

- Salton, G. (1971). *The SMART Retrieval System - Experiments in Automatic Document Processing*. Prentice-Hall.
- Santos, G., Santos, K., Valente, M. T., Serey, D., and Anquetil, N. (2013). Topicviewer: Evaluating remodularizations using semantic clustering. In *IV Congresso Brasileiro de Software: Teoria e Prática (Sessão de Ferramentas)*, pages 1–6.
- Santos, G., Valente, M. T., and Anquetil, N. (2014). Remodularization analysis using semantic clustering. In *1st CSMR-WCRE Software Evolution Week*, pages 224–233.
- Santos, K., Guerrero, D., Figueiredo, J., and Bittencourt, R. (2012). Towards a prediction model for source code vocabulary. In *1st International Workshop on the Next Five Years of Text Analysis in Software Maintenance*, pages 1–5.
- Sarkar, S., Ramachandran, S., Kumar, G. S., Iyengar, M. K., Rangarajan, K., and Sivagnanam, S. (2009). Modularization of a large-scale business application: A case study. *IEEE Software*, 26(2):28–35.
- Scanniello, G., D’Amico, A., D’Amico, C., and D’Amico, T. (2010). Using the kleinberg algorithm and vector space model for software system clustering. In *18th International Conference on Program Comprehension*, pages 180–189.
- Silva, L., Valente, M. T., and Maia, M. (2014). Assessing modularity using co-change clusters. In *13th International Conference on Modularity*, pages 1–12.
- Sindhgatta, R. and Pooloth, K. (2007). Identifying software decompositions by applying transaction clustering on source code. In *31st Annual International Computer Software and Applications Conference*, pages 317–326.
- Soares, G., Gheyi, R., and Massoni, T. (2013). Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, pages 147–162.
- Stevens, W. P., Myers, G. J., and Constantine, L. L. (1974). Structured design. *IBM Systems Journal*, 38(3):115–139.
- Taube-Schock, C., Walker, R. J., and Witten, I. H. (2011). Can we avoid high coupling? In *25th European conference on Object-oriented programming*, pages 204–228.
- Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). Qualitas corpus: A curated collection of java code for empirical studies. In *17th Asia Pacific Software Engineering Conference*, pages 336–345.

- Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. S. (2013). Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *ACM SIGSOFT Software Engineering Notes*, 38(5):1–4.
- Terra, R. and Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Software, Practice and Experience*, 39(12):1073–1094.
- Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2012). Recommending refactorings to reverse software architecture erosion. In *16th European Conference on Software Maintenance and Reengineering*, pages 335–340.
- Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2014). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, pages 1–36.
- Ujhazi, B., Ferenc, R., Poshyvanyk, D., and Gyimothy, T. (2010). New conceptual coupling and cohesion metrics for object-oriented systems. In *10th Working Conference on Source Code Analysis and Manipulation*, pages 33–42.
- Vakilian, M., Chen, N., Negara, S., Rajkumar, B. A., Bailey, B. P., and Johnson, R. E. (2012). Use, disuse, and misuse of automated refactorings. In *34th International Conference on Software Engineering*, pages 233–243.
- van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., and Riva, C. (2004). Symphony: View-driven software architecture reconstruction. In *4th Working Conference on Software Architecture*, pages 122–134.
- Verbaere, M., Godfrey, M. W., and Girba, T. (2008). Query technologies and applications for program comprehension. In *16th International Conference on Program Comprehension*, pages 285–288.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers.
- Wong, W. E., Gokhale, S. S., and Horgan, J. R. (2000). Quantifying the closeness between program components and features. *Journal of Systems and Software*, pages 87–98.

Zimmermann, T., Diehl, S., and Zeller, A. (2003). How history justifies system architecture (or not). In *6th International Workshop on Principles of Software Evolution*, pages 73–83.

Zipf, G. K. (1932). *Selected Studies of the Principle of Relative Frequency in Language*. Harvard University Press.