



**Universidade Federal da Bahia
Universidade Salvador
Universidade Estadual de Feira de Santana**

TESE DE DOUTORADO

**Understanding Software Cohesion Metrics: Experimental
Assessment of Conceptual Cohesion**

Bruno Carreiro da Silva

**Programa Multiinstitucional de
Pós-Graduação em Ciência da Computação – PMCC**

Salvador
2015

PMCC-Dsc-0021

BRUNO CARREIRO DA SILVA

**UNDERSTANDING SOFTWARE COHESION METRICS:
EXPERIMENTAL ASSESSMENT OF CONCEPTUAL COHESION**

Tese apresentada ao Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Estadual de Feira de Santana e Universidade Salvador, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Cláudio Nogueira Sant'Anna
Co-orientadora: Christina von Flach Garcia Chavez

Salvador
2015

Silva, Bruno Carreiro da

Understanding Software Cohesion Metrics: Experimental Assessment of Conceptual Cohesion / Bruno Carreiro da Silva. – 2015.

184p.: il.

Inclui apêndices.

Orientador: Prof. Dr. Cláudio Nogueira Sant'Anna.

Co-orientadora: Prof^a. Dr^a. Christina von Flach Garcia Chavez.

Tese (doutorado) – Universidade Federal da Bahia, Instituto de Matemática, Universidade Salvador, Universidade Estadual de Feira de Santana, 2015.

1. Engenharia de Software. 2. Coesão de Software. 3. Métricas de Software.
4. Evolução de Software.

I. Sant'Anna, Cláudio. II. Chavez, Christina. III. Universidade Federal da Bahia, Instituto de Matemática. IV. Universidade Salvador. V. Universidade Estadual de Feira de Santana. VI. Título.

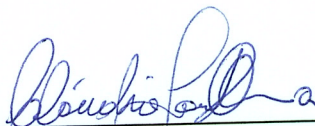
CDD – 005.1

CDU – 004.41

"UNDERSTANDING SOFTWARE COHESION METRICS: EXPERIMENTAL
ASSESSMENT OF CONCEPTUAL COHESION"

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multi-institucional de Pós-Graduação em Ciência da Computação da UFBA-UEFS-UNIFACS.

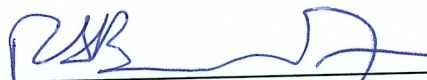
Salvador, 03 de agosto de 2015.



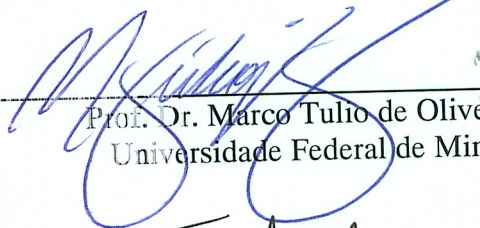
Prof. Dr. Cláudio Nogueira Sant'Anna
Universidade Federal da Bahia



Prof. Dr. Manoel Gomes de Mendonça Neto
Universidade Federal da Bahia



Prof. Dr. Roberto Almeida Bittencourt
Universidade Estadual de Feira de Santana



Prof. Dr. Marco Tulio de Oliveira Valente
Universidade Federal de Minas Gerais



Prof. Dr. Tiago Lima Massoni
Universidade Federal de Campina Grande

RESUMO

Coesão tem sido reconhecida como um importante atributo de qualidade em design de software ao longo de décadas. Coesão é definida como o grau em que um módulo está focado em um único interesse do software. Entretanto, medir coesão não é trivial, pois é difícil capturar os interesses realizados por módulos de software. Diversas métricas de coesão tem sido propostas nas últimas décadas e incorporadas em ferramentas de medição de software em escala industrial. Métricas de coesão estrutural centram-se no grau em que elementos internos de um módulo são estruturalmente relacionados uns com os outros. Coesão, entretanto, nem sempre é bem representada pela estrutura interna do módulo. Por exemplo, em programas orientados a objetos, é possível encontrar classes que implementam apenas um interesse, mas possuem métodos que raramente chamam uns aos outros ou que não acessam atributos em comum. Conhecendo esta limitação, pesquisadores têm proposto uma forma alternativa de medir coesão: as chamadas métricas conceituais de coesão. Tais métricas baseiam-se na identificação explícita de interesses realizados por cada módulo. Entretanto, há uma falta de entendimento sobre como métricas conceituais de coesão se encaixam entre diversas métricas tradicionais de coesão estrutural.

Neste contexto, nós realizamos uma série de estudos empíricos a fim de avaliar como e em que extensão métricas de coesão conceitual diferenciam-se das métricas tradicionais de coesão estrutural. De modo geral, nossa pesquisa envolveu: (i) um web-survey com oitenta desenvolvedores de software de nove países; e (ii) o repositório de seis sistemas de médio a grande porte, com código aberto e amplamente utilizados na indústria por diversos anos, incluindo um conjunto de aproximadamente trinta mil revisões no código fonte.

Como resultado, nós pudemos explicar que coesão conceitual é ortogonal a coesão estrutural e representa uma dimensão adicional na medição de coesão. Nós encontramos também que tal dimensão adicional é mais próxima de como desenvolvedores raciocinam sobre coesão de módulos. Adicionalmente, encontramos que coesão conceitual é um potencial indicador de propensão a mudança. Por fim, nós analisamos em que grau diferentes estratégias de mapeamento de interesses em código fonte impactam nos resultados de medição de coesão conceitual. Resultados mostraram que, quando viável, o mapeamento manual deve ser escolhido. Entretanto, como ele requer esforço elevado, uma das estratégias automáticas investigadas pode também ser considerada quando o mapeamento manual não for viável.

Nós disponibilizamos na web todos os nossos dados e materiais de estudo, incluindo o mapeamento manual de interesses que nós produzimos para um sistema real, e três extensões de ferramenta para computar uma métrica de coesão conceitual. Esses materiais podem ser utilizados ou estendidos por pesquisadores em estudos futuros. Por fim, nossos resultados podem ser diretamente usados por engenheiros de software quando estiverem

planejando ou executando medição de coesão em suas tarefas e quando estiverem construindo ferramentas de medição para ambientes de desenvolvimento. De forma geral, isto justifica esforço adicional para continuar melhorando as tecnologias e o corpo de conhecimento sobre medição de coesão.

Palavras-chave: Coesão de software; Métricas de coesão conceitual; Métricas de coesão estrutural; Propensão à mudança; Engenharia de software experimental

ABSTRACT

Cohesion has been recognized as an important quality attribute of software design across decades. It is defined as the degree to which a module is focused on a single concern. However, measuring cohesion is not trivial as it is difficult to capture the concerns addressed by software modules. Several structural cohesion metrics have been proposed in the last decades and incorporated in industry-scale software measurement tools. Structural cohesion metrics rely on the degree to which the internal elements of a module are structurally related to each other. Cohesion, however, is not always well represented by module internal structure. For instance, in object-oriented programs, we can find classes that implement just one concern but have methods that seldom call each other or do not access attributes in common. Knowing this limitation, researchers have recently proposed an alternative way of measuring cohesion: the so-called conceptual cohesion metrics. Conceptual cohesion metrics are based on the explicit identification of the concerns realized by each module. However, there is a lack of understanding on how conceptual cohesion metrics fit among a number of conventional structural cohesion metrics.

In this context, we performed a series of empirical studies in order to evaluate how and in which degree conceptual cohesion metrics differ from conventional structural cohesion metrics. Overall, our research was based on data obtained from: (i) a web-survey with eighty software developers from nine countries and (ii) the repository of six open source, long-lived, industry-scale, medium to large-sized systems with a change set of around thirty thousand source code revisions.

As a result, we could explain that conceptual cohesion is orthogonal to structural cohesion and represents an additional dimension of cohesion measurement. We also found that such additional dimension is closer to how developers reason about module cohesion. In addition, we found that conceptual cohesion is a potential indicator of module change-proneness. Finally, we analyzed in which degree different strategies for mapping concerns to source code impact on conceptual cohesion measurement results. The results showed that, when feasible, manual mapping should be chosen. However, as it requires much effort, one of the investigated automatic mapping strategies can be also considered when the manual strategy is not feasible.

We made all of our study datasets and materials public available on the web, including the concern mapping that we manually produced for a real system, and three tool extensions for computing a conceptual cohesion metric. These materials can be used or extended by researchers in future studies. In summary, our results can be directly used by software engineers when planning and executing cohesion measurement in their tasks and when building metric tools for development environments. Overall, this justifies additional effort to keep improving cohesion measurement body of knowledge and technology.

Keywords: Software cohesion; Conceptual cohesion metrics; Structural cohesion metrics; Change-proneness; Empirical software engineering

CONTENTS

Chapter 1—Introduction	1
1.1 Problem Statement	2
1.2 Main Goal and Research Questions	6
1.3 Contributions	9
1.4 Publications	10
1.5 Thesis Outline	10
Chapter 2—Different Flavors of Cohesion Metrics	13
2.1 Structural Cohesion Metrics	13
2.1.1 Explaining the Main Structural Cohesion Metrics	14
2.1.2 Metrics Comparison	17
2.2 Conceptual Cohesion Metrics	19
2.2.1 Text Mining-based Cohesion Metrics	19
2.2.2 The Lack of Concern-based Cohesion Metric	22
2.2.3 Software Concerns and Concern Mapping	23
2.2.4 Summarizing Conceptual Cohesion Metrics	26
2.3 Summary	27
Chapter 3—Unveiling a Hidden Dimension of Cohesion Measurement	29
3.1 Study Settings	30
3.1.1 Selected Systems	30
3.1.2 Conceptual Cohesion Measurement Procedure	31
3.1.2.1 Measuring LCbC using XScan	31
3.1.2.2 Measuring MWE using TopicXP	32
3.1.3 Structural Cohesion Measurement	34
3.1.4 Method of Analysis	34
3.2 Study Results and Discussion	35
3.2.1 Unveiling Conceptual Cohesion Dimensions	35
3.3 Threats to Validity	38
3.4 Related Work	39
3.5 Summary	40

Chapter 4—How Developers reason about Module Cohesion	43
4.1 Related Work	44
4.2 Study Design	45
4.2.1 Survey Overall Structure	45
4.2.2 Comparing and Rating Cohesion between two Classes	46
4.2.2.1 Class Selection Criteria for Cohesion Comparison	47
4.2.2.2 Cohesion Metrics	48
4.2.2.3 Selected Pairs of Classes	49
4.2.3 Collecting Participant’s Profile	50
4.2.4 Survey Sampling	50
4.2.5 Data Collection	51
4.3 Results and Analysis	51
4.3.1 Participants’ Profile	52
4.3.2 How do developers perceive module cohesion? How do they reason about it?	53
4.3.2.1 “Are you familiar with the concept of cohesion in the con- text of software development?”	54
4.3.2.2 “How would you explain to someone else what a highly cohesive class is?”	55
4.3.2.3 Rationale for Rating and Comparing Cohesion	57
4.3.3 To what extent do structural cohesion and conceptual cohesion re- late with how developers rate cohesion?	58
4.3.3.1 Cohesion Ratings and Respective Rationale	58
4.3.3.2 Cohesion Ratings vs. Cohesion Familiarity	62
4.3.3.3 Cohesion Ratings vs. Participants’ Experience	63
4.4 Threats to Validity	65
4.5 Summary	66
Chapter 5—Conceptual Cohesion and Change-Proneness	69
5.1 Study Settings	70
5.1.1 Change Counting	71
5.1.2 Method of Analysis	72
5.2 Results and Discussion	73
5.3 Threats to Validity	76
5.4 Related Work	77
5.5 Summary	78
Chapter 6—The Impact of Different Concern Mapping Strategies on Conceptual Cohesion	79
6.1 Study Settings	80
6.1.1 Analyzed Systems	81
6.1.2 Automatic Mapping Strategies	82

6.1.3	Manual Mapping Strategy	83
6.1.4	Change Count Measurement	86
6.2	Results and Analysis	87
6.2.1	Concern Mappings and LCbC Distributions	87
6.2.1.1	Auxiliary Artifacts Drive the Accuracy of Manual Mappings	87
6.2.1.2	Less Coverage of XScan Mappings	90
6.2.1.3	Low LCbC Variance of Topic-based Mappings	90
6.2.1.4	Statistical Tests	91
6.2.1.5	Characterizing XScan Mapping Results	92
6.2.1.6	Characterizing Topic-based Mapping Results	92
6.2.1.7	XScan Requires less Effort	93
6.2.1.8	They are different but are they correlated with each other?	93
6.2.2	Correlation between LCbC and Change-Proneness	95
6.2.2.1	Mapping Size Does not Matter	95
6.2.2.2	XScan Mapping when Manual Mapping is not Possible .	96
6.3	Threats to Validity	96
6.4	Related Work	98
6.5	Summary	99
Chapter 7—Conclusion		101
7.1	Future Research Directions	102
Appendices		115
Appendix A—R Script for PCA		115
Appendix B—Survey Questionnaire		119
Appendix C—Survey Classes		125
Appendix D—Data Transformation and Application of Fleiss Kappa Test		149
Appendix E—Application of Fisher Exact Test		153
Appendix F—R Scripts for Correlation Tests and Regression Trees		163
Appendix G—Regression Trees		167
Appendix H—Application of Friedman Test		175

Appendix I—jEdit Set of Concerns

179

LIST OF FIGURES

1.1	A sample class and its internal related elements	3
1.2	A sample class and its additional behaviors exposed	4
1.3	A sample class and its internal related elements	5
2.1	A sample class with mapped concerns	23
2.2	Illustrative concern mapping relationship	24
3.1	Principal components of each system	36
4.1	Questions flow overview.	46
4.2	Classes for comparison.	49
4.3	Participants' occupation and academic degree.	53
4.4	Participants' self estimation on programming experience.	54
4.5	Participants' self estimation on Java and OOP.	54
4.6	Participants' programming experience in years.	55
4.7	Coded topics from how participants explain highly cohesive classes (survey first question).	56
4.8	Cohesion ratings for the three scenarios.	59
5.1	An example of a regression tree having CC as response variable	73
5.2	Correlation between each cohesion metric and change count	74
6.1	LCbC distributions using manual mapping strategy	89
6.2	LCbC distributions using XScan and Topic-based mapping strategies	91
B.1	Cohesion familiarity	120
B.2	Cohesion explanation	120
B.3	Information about cohesion definition	121
B.4	First scenario about cohesion comparison and rating	121
B.5	Second scenario about cohesion comparison and rating	122
B.6	Third scenario about cohesion comparison and rating	123
B.7	Participant profile	124
G.1	Tomcat regression tree.	168
G.2	Findbugs regression tree.	169
G.3	Freecol regression tree.	170
G.4	JFreeChart regression tree.	171
G.5	Rhino regression tree.	172

G.6 JEdit regression tree. 173

LIST OF TABLES

2.1	Conceptual cohesion metrics	27
3.1	Summary of the systems under study	30
3.2	Summary of XScan applied and corresponding LCbC distributions	32
3.3	LDA parameters applied and a summary of MWE distributions	33
4.1	Programming experience and coded topics for cohesion ratings explanation	64
5.1	Systems and analyzed change-set	71
6.1	Systems and change-set analyzed	82
6.2	A sample of jEdit set of concerns	84
6.3	Summary of Concern Mapping Strategies Applied	88
6.4	Correlation among LCbC variations for Rhino and jEdit	94
6.5	Correlation among LCbC variations for JFreeChart, Findbugs, Tomcat and Freecol	94
6.6	Correlation between Change Count and LCbC measurements	95

INTRODUCTION

The term *module cohesion* was first introduced by Stevens, Myers and Constantine (1974) in the context of structured design and defined as a measure of the degree to which the internal elements of a module belong together. Accordingly, in a highly cohesive module all the elements are related to the performance of a single concern of the software. A concern is generally interpreted as any property of interest for software project stakeholders. It is normally considered as a conceptual unit to be treated in a modular way (Tarr et al., 1999), (Robillard; Murphy, 2007), (Sant’Anna, 2008), (Figueiredo, 2009). Concerns can range from high-level properties such as quality of service and distributed communication to low-level properties such as data sorting and buffering. They can also represent functional properties like business rules and features, or non-functional such as transaction management and synchronization.

Since Stevens, Myers and Constantine’s work, other authors have studied the property of module cohesion and gave similar definitions. For instance, Yourdon and Constantine (1979) stated that cohesion is “the degree of functional relatedness of processing elements within a single module”. Bieman and Kang (1995) add to the classical definitions stating that a highly cohesive module should be difficult to split since its internal elements are closely related to each other.

Martin (2003) defined cohesion as the Single Responsibility Principle¹ - “There should never be more than one reason for a class to change”. Generally, the more concerns a class addresses the more reasons to undergo changes. According to Martin’s view, a highly cohesive class should realize a single concern, and that concern should be entirely encapsulated by the class. All its services should be narrowly aligned with that concern.

Regardless the several ways of defining module cohesion, different authors pointed cohesion as a desired quality attribute in software engineering (Myers, 1978; Yourdon; Constantine, 1979; Bieman; Kang, 1995; Fenton; Pfleeger, 1998; Martin, 2003; Marcus; Poshyvanyk; Ferenc, 2008). For example, some authors argue that during the development process developers may apply the concept of cohesion to reason on whether or

¹In this text, *responsibility* is a synonym of *concern*. Martin’s principle can also be read as *Single Concern Principle*.

not, and at what extent, they are designing and coding modular components. Managers may also use the concept of cohesion in order to assess the quality of the design and code artifacts and to support them in making decisions about effort investment, cost reduction, task planning and so on. Across decades researchers have claimed that highly cohesive modules are easier to understand and evolve than less cohesive modules (Briand; Bunse; Daly, 2001; Chen et al., 2002; Marcus; Poshyvanyk; Ferenc, 2008; Dallal; Briand, 2012). Therefore, the applicability of cohesion measurement may play an important role in software engineering activities.

1.1 PROBLEM STATEMENT

Measuring cohesion is not straightforward. Indeed, several researchers have attempted to provide an objective and effective way to measure cohesion (Chapter 2). Defined metrics rely on structural information extracted from object-oriented source code, for example, pairs of methods of a class that access the same attributes.

Figure 1.1 shows a sample Java class with two attributes and nine methods, which is part of a system that builds and represents family relationships as trees. Particularly, this class represents the spouse relationship between two people – the *husband* and *wife* attributes. The methods involve: getting access to the private attributes, getting the relation type or the partner object given a person as parameter, objects comparison, and export information of the relationship to a given file. The thin arrow indicates the methods that access the *husband* attribute, while the thick arrow indicates the methods that access the *wife* attribute.

The underlying idea of most of the structural cohesion metrics is to quantify the degree to which methods belong together by considering if they share attributes or not. The more a class has methods using the same attributes, the more cohesive the class is. If a class has more methods using distinct attributes than methods sharing attributes, it is interpreted as a class with more than one focus, that is, its methods are not related to the performance of a single concern of the software. Then, the class is said to lack cohesion. That is the exact reasoning behind the well-know cohesion metric called Lack of Cohesion in Methods (LCOM) (Chidamber; Kemerer, 1994). LCOM is an inverse metric, as it measures the lack of cohesion. Thus, the higher the LCOM value, the lower is the cohesion degree. In the example of Figure 1.1, LCOM yields zero (high cohesion), because there are more pair of methods sharing attributes than pair of methods not sharing attributes. Similarly, other two structural cohesion metrics indicate that this class is highly cohesive: TCC (Tight Class Cohesion) (Bieman; Kang, 1995) and LCOM4 a variation of LCOM by (Hitz; Montazeri, 1995). According to LCOM4 that class yields 0.2 in a range of 0 (low lack of cohesion) to 1 (high lack of cohesion), thus also indicating a highly cohesive class. According to TCC, that class yields 0.8 in a range of 0 (low cohesion) to 1 (high cohesion). A more detailed explanation about the main structural cohesion metrics available in literature and tools is presented in Chapter 2.

Such structural point of view is a reasonable way for measuring cohesion. In theory, when a problem is decomposed into modules – in our case, classes – it is expected to have classes encapsulating attributes and methods which are supposed to be related to

```

public class RelationSpouse extends Relation {
    > private Person husband;
    > private Person wife;
    -public RelationSpouse (Person husband, Person wife) {}
    -public String getRelationType(Person person) {}
    -public Person getPartner(Person person) {}
    public Person getHusband() {}
    -public Person getWife() {}
    -public Person getPerson1() {}
    public Person getPerson2() {}
    -public boolean equals(Object o) {}
    -public void printToFile(File f) throws IOException {}
}

```

Figure 1.1 A sample class and its internal related elements

each other in order to perform a well-defined cohesive behavior, i.e. focusing on a single concern. Inversely, when a class is focused on more than one concern, it is expected to have different groups of methods and attributes related to their respective concerns. And in this latter configuration the class is likely to be split. Additionally, this type of metrics is inexpensive to compute. Once the source code is available, nothing else is needed to measure the cohesion degree.

However, this notion of cohesion is too dependent on the source code structure and does not consider any abstract information regarding the concerns implemented by the classes. Most of the cohesion metrics are not capable to capture the actual concerns of a module regardless of its internal syntactic structure. No matter whether or not the methods and attributes of a class are structurally related to each other, it should be considered highly cohesive if it represents a crisp abstraction of a software concern. For instance, in the example of Figure 1.1, although the class is considered to be cohesive according to its structure, it has different concerns contributing to the implementation of more than one software behavior. Figure 1.2 shows the same class but now highlighting with gray boxes the three code fragments implementing two secondary concerns in addition to the primary one. Therefore, besides what the class is supposed to implement (i.e. representing spouse relationship), the first two boxes indicate the invocation of a window to output a message in the user interface. The third box indicates a method dedicated to export the relationship represented by the class to a given file. Therefore, from that perspective, this class is not as cohesive as it seems to be according to struc-

tural cohesion way of measurement. This class implements software concerns which could be encapsulated in other modules, and according to Martin's view (Martin, 2003) it has more than one reason to change. Depending on the software size, the presence of such lack of cohesion may negatively impact on understandability and maintainability.

```

public class RelationSpouse extends Relation {
    public Person husband;
    public Person wife;

    public RelationSpouse (Person husband, Person wife){

    public String getRelationType(Person person){
        if (person.equals(husband))
            return WIFE;
        else if (person.equals(wife))
            return HUSBAND;
        else{
            JOptionPane.showMessageDialog(null,
                "Wrong person type for spouse relationship..."); output
            return INVALID_RELATIONSHIP;
        }
    }

    public Person getPartner(Person person){
        if (person.equals(husband))
            return wife;
        else if (person.equals(wife))
            return husband;
        else{
            JOptionPane.showMessageDialog(null,
                "Wrong person type for spouse relationship..."); output
            return null;
        }
    }

    public Person getHusband() {
    public Person getWife() {
    public Person getPerson1() {
    public Person getPerson2() {
    public boolean equals(Object obj) {
    public void printToFile(File f) throws IOException{
    }
}

```

Figure 1.2 A sample class and its additional behaviors exposed

Figure 1.3 shows another example where a structural cohesion metric does not succeed on indicating the degree to which a class is focused on a single or few concerns. In this example we took a Java class from a hotel management system available at SourceForge repository¹. This class is designed to handle database connections. It is entirely focused on opening and closing connections to a given database instance, and managing a connection pool. Therefore, this class is highly cohesive as it is focused on the concern of *handling database connections*. However, according to LCOM definition by (Henderson-sellers, 1996) (also known as LCOM5) this class lacks cohesion. Its LCOM5 value is 0.8 from

¹<http://sourceforge.net/projects/fgmp-hm/>

a range of 0 (low lack of cohesion) to 1 (high lack of cohesion). The arrows in Figure 1.3 indicate methods accessing class attributes. Details about how LCOM5 computes cohesion can be found in Chapter 2. In summary, LCOM5 yields a high value because this class has distinct groups of methods not accessing attributes in common. For instance, the attributes *Host*, *Port*, *Database*, *User*, and *Password* are used by only one half of the methods; while *con_pool* is used by the other half; Also, the *concount* attribute is used only by one method.

```

public class DB_Backend {
    > private final static int conns= 5; <
    > private static int concount=-1;
    > public static Connection [] con_pool = new Connection [conns]; <
    {
        private String Host;
        > private String Port;
        private String Database;
        private String User;
        private String Password;
    }
    --- public DB_Backend(String h, String p, String d, String u, String pw) {}
    --- public void init(String h, String p, String d, String u, String pw) {}
    --- public Boolean connect_DB() {}
    --- public static Boolean connect_DB(String host, String database, String user, String PW) {}
    --- public static Connection getConnection() {}
    --- public static void close_DB() {}
}

```

Figure 1.3 A sample class and its internal related elements

The examples illustrated in Figures 1.1, 1.2 and 1.3 show that there might be situations where structural cohesion metrics fail to apply the cohesion concept as the internal structure of a module not always represents well the amount of concerns the module implements. More recently, researchers have proposed alternative ways of measuring cohesion which try to capture information about what concerns a module internally addresses (Marcus; Poshyvanyk, 2005), (Liu et al., 2009), (Ujhazi et al., 2010), (Silva et al., 2012). For this purpose, it is necessary to build a *concern mapping* for computing this kind of cohesion metrics. A concern mapping is the assignment of software concerns to source code elements (e.g. attributes, methods, classes).

For example, the MWE (Maximal Weighted Entropy) metric (Liu et al., 2009) uses a concern mapping automatically generated from a text-mining technique. It extracts the concerns involved in a system by processing all the comments and identifiers as words. The distribution of words across the system modules gives information of what concerns each module implements. For instance, the words [host, port, database, user, password, connection] could represent the *database connection handling* concern mapped to the class of Figure 1.3. It could be identified by mining these words from the entire system

and evaluating that they are concentrated in *DB_Backend* class.

Another example is the Lack of Concern-based Cohesion (LCbC) metric (Silva et al., 2012), which considers the lack of cohesion as the number of concerns a module addresses. This metric does not define a specific concern mapping technique to be applied. Thus, it can be used with a text mining-based concern mapping, like MWE, or with any other chosen technique. For instance, we could manually build a concern mapping assigning the database handling concern to the entire *DB_Backend* class. Then, LCbC value would be 1. In the example of Figure 1.2, we manually mapped three concerns for that class, so LCbC yields 3. Therefore, with respect to LCbC, *RelationSpouse* is less cohesive than *DB_Backend*.

These metrics form the group of *conceptual cohesion* metrics. They are called *conceptual* as the first metrics of this kind were described based on the term *concept*, which has the same meaning of *concern*. Here, we prefer to use the term concern instead of concept. In Chapter 2 we explain conceptual cohesion metrics in more details.

Nevertheless, there is a lack of empirical evidence about the applicability of conceptual cohesion measurement in software product assessment. In theory, the consideration of module concerns may put conceptual cohesion closer to the human-oriented view of cohesion. Presumably, it is easier for developers, when decomposing a problem into modules, to find different concerns implemented by a module than to calculate the degree of structural relatedness amongst the methods and attributes of the module. However, there is no empirical evidence to support this argument. Also, we do not know whether or not the approach of measuring cohesion through concern mappings adds to or overlaps conventional structural cohesion measurement.

1.2 MAIN GOAL AND RESEARCH QUESTIONS

The main goal of this research is to understand conceptual cohesion by investigating through empirical studies how it performs as an alternative way of cohesion measurement when compared with conventional structural cohesion. Therefore, our overarching research question is:

What does explain conceptual cohesion as a different way of cohesion measurement in contrast to conventional structural cohesion?

Accordingly, we defined a set of more specific research questions that guided the empirical studies we present in this research. In this section we define specific research questions as well as the reasoning and assumptions that motivate them.

There are several cohesion metrics available in the literature (Chapter 2). Most of them are conventional structural cohesion metrics. They have been target of most of the studies in the last decades. Also, they have been implemented in several open source and commercial metric tools. So a substantial knowledge about cohesion measurement and its applicability was built from investigations on structural cohesion measurement. However, conceptual cohesion is based on a different source of information and counting mechanism. Therefore, it is necessary to understand through empirical evidence if that difference leads to significantly different results on cohesion measurement. In other words we should assess whether conceptual cohesion confirms itself as a distinct approach of

measuring cohesion not only in terms of counting mechanism but also in terms of generated results. This should give us scientific evidence that it can be further investigated as opposite of being another metric that measures cohesion and generate results in the same way others already do. In that context, our first research question is:

RQ1. Does the conceptual nature of conceptual cohesion metrics make them significantly different from structural cohesion metrics?

Furthermore, we hypothesize that conceptual cohesion is closer to the human-oriented view of cohesion. Software engineers would prefer cohesion metrics that allow them to reason about a module as a set of concerns related to the software instead of a set of code segments that are syntactically associated to each other (Etzkorn; Delugach, 2000). However, empirical evidence must be drawn from this hypothesis. Basically, we need first to understand what the developers' opinion about module cohesion are and how they reason about it. Then, we can evaluate whether or not conceptual cohesion is closer to the human view of cohesion when compared with conventional structural cohesion. That leads us to the following research questions:

RQ2. How do developers perceive module cohesion? And how do they reason about it?

RQ3. To what extent do structural cohesion and conceptual cohesion measurements relate with how developers rate cohesion of modules?

In *RQ2* we aim at understanding first what developers know about cohesion and how they perceive that concept when: (i) asked to define what cohesive modules are; and (ii) when they have the task to compare the cohesion degree between two modules. To the best of our knowledge, there is no available studies in literature investigating what the actual developers concept about module cohesion is and how they use that concept when comparing the cohesion of modules.

In *RQ3*, the intention is to analyze and compare the relation between structural and conceptual ways of cohesion measurement with how developers rate module cohesion. We have the hypothesis that, in general, conceptual cohesion is closer to developers rating of cohesion than structural cohesion, because it focus on the abstraction of modules concerns. In theory, such raising of abstraction level (from code structure to concerns implementation) for measuring cohesion leads to a cohesion measurement closer to how developers work. Development tasks are usually related to the implementation of new features, fixing faults, adapting modules to receive new requirements and so on. This discussion was already raised in the literature (Etzkorn; Delugach, 2000), but, again, there is no evidence to support this hypothesis.

After addressing those initial questions we could gather empirical evidence that conceptual cohesion is worth to be further investigated. We could empirically validate that conceptual cohesion is orthogonal to structural cohesion in a way that it captures additional dimension of cohesion measurement. Also, we found that conceptual cohesion

is closer to how developers reason about module cohesion even for unexperienced developers. However, those partial results do not point out whether or not conceptual cohesion may associate to software quality attributes. Software product metrics such as coupling, size and cohesion are often empirically assessed in terms of their impact on quality attributes such as fault-proneness, comprehension or maintenance effort, and change-proneness (Basili; Briand; Melo, 1996). The intention is to understand whether and to what extent a metric has the potential to be an indicator of a software quality attribute. There is a lack of evidence about whether conceptual cohesion could possibly complement or replace conventional structural cohesion metrics in the role of indicating different degrees of a given quality attribute.

In this context, we aim at investigating the well known Martin's view (2003) of module cohesion. He claims that a class addressing several concerns lacks cohesion and therefore it is more likely to undergo changes. Whereas a highly cohesive class should focus on a single concern decreasing its likelihood to undergo changes. Accordingly, as conceptual cohesion metrics rely on classes' concerns, we hypothesize that this way of measuring cohesion has a potential association to class change-proneness. Change-proneness is a software quality attribute (Madhavji; Fernandez-ramil; Perry, 2006) that refers to the degree to which a software module is likely to change along its evolution history. Knowing which modules are more likely to change over time can improve the process of developing and evolving new or existing software modules in a given project by focusing attention on them. For instance, managers can drive resources on the least cohesive modules in order to improve their cohesion and then reduce effort on future changes. Developers can have the cohesion degree of modules in mind when design new modules or maintaining existing ones, which may also reduce effort on future changes. Therefore, this led us to the fourth research question:

RQ4. Whether and to what extent does conceptual cohesion associate to change-proneness?

When addressing RQ4 we could find that conceptual cohesion is a potential change-proneness indicator in place of or complementing structural cohesion in this role. However, with the empirical studies we carried out for answering the four initial questions we could observe that conceptual cohesion metrics are very sensitive to how software concerns are mapped to source code elements. There are several different strategies for that purpose that can be used to support conceptual cohesion measurement. According to our research, there is not any study that has explored the impact of varying strategies for concern mapping on conceptual cohesion measurement. For those reasons we stated the following questions:

RQ5. Do different strategies for mapping module concerns in source code impact on conceptual cohesion measurement?

RQ6. If such impact is significant, can we explain how different is conceptual cohesion over different strategies?

RQ7. Does it influence on the association between conceptual cohesion and module

change proneness?

Therefore, we could provide empirical results to better explain the influence of different concerns mapping strategies on conceptual cohesion measurement and on the role of conceptual cohesion as a change-proneness indicator.

1.3 CONTRIBUTIONS

We present the following research contributions as result of a series of empirical studies guided by the research questions previously described:

1. Empirical validation through statistical assessment that conceptual cohesion is orthogonal to structural cohesion. We could demonstrate that conceptual cohesion measurement captures an additional dimension of module cohesion which is not captured by well-known conventional structural cohesion metrics.
2. Qualitative assessment with statistical support to explain how developers reason about module cohesion. We could provide empirical evidence that developers usually perceive cohesion based on what concerns a given module addresses. Also, we demonstrated that developers' ratings about module cohesion are closer to conceptual cohesion than structural cohesion.
3. Empirical validation of conceptual cohesion metrics as potential change-proneness indicators. We also explained typical situations when conceptual cohesion succeeds or fails in playing the role of change-proneness indicator in comparison to structural cohesion.
4. A thorough analysis with statistical support to better understand and explain the impact of different concern mapping strategies on conceptual cohesion measurement and their influence on how conceptual cohesion associates to change-proneness.
5. A new manual concern mapping over jEdit source code made available and open for research community, as explained in Chapter 6. There is a lack of large-scale long-lived software projects with a rich change set having a manual concern mapping made by developers or researchers with the intent to find as many concerns as possible. This outcome has a potential to evolve and become a benchmark like the concerns of Rhino project (Eaddy et al., 2008b), which is the only one available with this setting.
6. Aggregate results from four empirical studies that indicate that conceptual cohesion is worth considering further effort for new empirical investigations and for technology improvement on software measurement tools.
7. Improved body of knowledge on cohesion measurement, specially regarding conceptual cohesion metrics applicability.
8. All material from the empirical studies were made publicly available on the web, so that they can be replicated or extended in further investigations.

9. Three tool extensions (ConcernTagger, XScan and TopicXP) for computing a conceptual cohesion metric called LCbC (explained in Chapter 2). These extensions were also made publicly available on the web.

1.4 PUBLICATIONS

Excerpts of this thesis have been published or submitted in journal, conference and workshop papers listed below in chronological order.

- Silva, B.; Sant'Anna, C.; Chavez, C. Concern-based Cohesion as Change-proneness Indicator: An Initial Empirical Study. In: *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*. Honolulu, Hawaii, USA. ACM, 2011, p. 52-58.
- Silva, B.; Sant'Anna, C.; Chavez, C.; Garcia, A. Concern-based Cohesion: Unveiling a Hidden Dimension of Cohesion Measurement. In: *Proceedings of the 20th IEEE International Conference on Program Comprehension*. Passau, Germany. IEEE Computer Society, 2012, p. 103-112.
- Silva, B.; Sant'Anna, C.; Chavez, C. An Empirical Study on How Developers Reason About Module Cohesion. In: *Proceedings of the 13th International Conference on Modularity*. Lugano, Switzerland. ACM, 2014, p. 121-132.
- Silva, B.; Sant'Anna, C.; Rocha, N.; Chavez, C. *The Impact of Different Concern Mapping Strategies on Conceptual Cohesion*. In: *Journal of Information and Software Technology*. Elsevier Science Inc., New York, 2015. (*under review*).

1.5 THESIS OUTLINE

In the rest of this document we elaborate on the issues outlined in the introduction. In **Chapter 2**, we present a review on the main structural cohesion metrics as well as all conceptual cohesion metrics we found in literature. It is followed by a comparative analysis between the metrics mentioned in that chapter.

In **Chapter 3** we present an empirical study to quantitatively assess whether conceptual cohesion metrics are different from conventional structural cohesion metrics. This includes an explanation about how we measured conceptual cohesion by using two distinct metrics and structural cohesion through a representative set of five metrics. We also elaborate discussion over some classes to exemplify what we found from the quantitative analysis.

In **Chapter 4** we describe our work on a survey to understand and explain how developers perceive module cohesion and how they rate cohesion between pairs of classes. We present all the survey structure and results as well as a discussion about our findings.

In **Chapter 5** we present an empirical assessment to analyze the association of cohesion metrics (conceptual and structural) with module change-proneness. This involves six long-lived, industry-scale, medium to large-sized systems. They had a rich change set scenario analyzed with a total of 30.248 revisions processed. In this chapter we also

discuss typical situations when conceptual cohesion succeeds or fails in indicating module change-proneness.

In **Chapter 6** we explore the impact of different concern mapping strategies on conceptual cohesion measurement. This includes two automatic strategies besides the manual one. Also, this chapter presents how we built a manual concern mapping over the jEdit system.

In **Chapter 7** we summarize our work by briefly describing what we have done in the context of this research as well as achieved contributions. In addition, we point out perspectives on future research directions.

DIFFERENT FLAVORS OF COHESION METRICS

This chapter explains a variety of cohesion metrics divided in two main sections. The first section focuses on presenting the main structural cohesion metrics available in literature and in software tools, whereas second section explains the conceptual cohesion metrics we found during our research. Overall, this chapter aims at giving an overview of a representative set of structural and conceptual cohesion metrics from which we selected the metrics applied in our empirical studies.

2.1 STRUCTURAL COHESION METRICS

The authors who first introduced the concept of module cohesion also described a qualitative scale for classifying the cohesion degree of software modules (Stevens; Myers; Constantine, 1974). That scale is defined from the lowest to the highest cohesion as: coincidental, logical, temporal, communicational, sequential and functional. This scale is not supposed to be linear and it neither gives a quantitative measure for cohesion, but it states different cohesion levels based on how the internal elements of a software module are related to each other. Further details can be found in (Stevens; Myers; Constantine, 1974). Although defined in the context of structured design it certainly influenced later approaches for measuring cohesion.

Years later, after the initial work by Stevens, Myers and Constantine, and with the emergence of Object-Oriented (OO) programming, analysis and design, several other authors have undertaken research on cohesion measurement for OO software modules. Several studies available in literature have provided comprehensive reviews about module cohesion for OO software (Briand; Daly; Wüst, 1998; Eitzkorn; Delugach, 2000; Badri; Badri; Gueye, 2008; Marcus; Poshyvanyk; Ferenc, 2008; Dallal; Briand, 2012). Amongst them, the unified framework for cohesion measurement by Briand et al. (1998) is the most complete analysis over the main cohesion metrics that had appeared until the end of the 90 decade. In the 1990s most of the well-known cohesion metrics for OO software have emerged.

In this section, we give an overview of the most traditional and most cited structural cohesion metrics presented in literature. It is not the purpose of this section covering all existing cohesion metrics which is certainly a long list. Rather, we aim at showing their main characteristics and understanding their underlying interpretation and counting mechanisms for measuring cohesion. It is also important to understand these structural cohesion metrics as some of them are used on empirical studies that are part of this research.

In Section 2.1.1 we explain ten structural cohesion metrics based on Briand et al.(1998) framework, as their work gives a comprehensive and critical review over the most well-known cohesion metrics based on a unified analysis and on pre-defined criteria. Some of the structural cohesion metrics presented here is also available in open-source and commercial tools for software measurement. In Section 2.1.2 we present a comparison among the ten metrics also based on Briand’s framework.

2.1.1 Explaining the Main Structural Cohesion Metrics

The first attempt for measuring cohesion in OO systems was the well-known Lack of Cohesion in Methods (LCOM) (Chidamber; Kemerer, 1991), hereafter called **LCOM1**¹. Chibamber and Kemerer defined cohesion of a class based on the degree of relatedness of its methods. For them, two methods are related to each other if they use at least one class attribute in common. Therefore, a class is formed by groups of methods, where each group has methods which are related to at least one other method of the group. The LCOM1 value for cohesion is the number of such groups.

Formally, consider a Class C_1 with methods M_1, M_2, \dots, M_n . Let $\{I_i\}$ be the set of attributes used by method M_i , there are n such sets $\{I_1\}, \dots, \{I_n\}$. Then, LCOM1 is the number of disjoint sets formed by the intersection of the n sets.

LCOM1 is an inverse measure, which means that a high value of LCOM1 indicates low cohesion and vice versa. This interpretation fits most of the cohesion metrics presented in this chapter.

After Chidamber and Kemerer’s seminal work (Chidamber; Kemerer, 1991), they and other authors presented other ways for measuring cohesion, basically extending or refining the LCOM1 definition.

The **LCOM2** metric refers to the second work of Chibamber and Kemerer where they refined their original definition (Chidamber; Kemerer, 1994). LCOM2 is calculated by the difference of two numbers: the number of pairs of methods in a class having no common attribute references and the number of pairs of methods that have at least one attribute reference in common. If such difference is negative, LCOM2 is set to zero. The following definition formalizes the metric.

Consider a Class C_1 with methods M_1, M_2, \dots, M_n . Let $\{I_i\}$ be the set of attributes used by method M_i , there are n such sets $\{I_1\}, \dots, \{I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. If all n sets $\{I_1\}, \dots, \{I_n\}$ are \emptyset then let $P = \emptyset$.

¹For the LCOM variations as well other cohesion metrics presented here we kept the same acronyms used in (Briand; Daly; Wüst, 1998).

$$LCOM2 = \begin{cases} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{otherwise} \end{cases}$$

Hitz and Montazeri (1995) used graph theory to define a new cohesion metric, called here as **LCOM3**. They take methods as vertices and edges linking two vertices if the corresponding methods have at least one attribute reference in common. Formally, let X denote a class, I_X the set of its attributes, and M_X the set of its methods. Consider a simple, undirected graph $G_X(V, E)$ with $V = M_X$ and $E = \{(m, n) \in V \times V \mid \exists i \in I_X : (m \text{ accesses } i) \wedge (n \text{ accesses } i)\}$. LCOM3 is then defined as the number of connected components of G_X .

LCOM3 is very similar to LCOM1, except that the former is more precisely defined by means of graph theory. Also, Hitz and Montazeri identified a problem with accessor methods for LCOM3. An accessor method provides read or write access to an attribute of the class. And by convention they typically reference only one attribute (the one they provide access to). The presence of accessor methods artificially decreases the class cohesion as measured by LCOM3.

To eliminate the problem with accessor methods, Hitz and Montazeri propose a second version of their LCOM metric - called here as **LCOM4**. This metric is actually an expansion of LCOM3 to include an edge between vertices representing methods m_1 and m_2 , if m_1 calls m_2 or vice versa. So, LCOM4 expands LCOM3 definition of the E set as follows: $E = \{(m, n) \in V \times V \mid (\exists i \in I_X : (m \text{ accesses } i) \wedge (n \text{ accesses } i)) \vee (m \text{ calls } n) \vee (n \text{ calls } m)\}$. According to LCOM4 any two methods are related if they access the same attribute and if at least one of them calls the other. As a consequence, whenever a class has methods indirectly accessing internal attributes by calling accessors its cohesion is not artificially decreased, as LCOM4 recognizes method relations by means of method calls.

Besides LCOM3 and LCOM4, Hitz and Montazeri defined the **Co** (“Connectivity”)² metric to discriminate different levels of cohesion when there is only one connected component in the graph (LCOM4=1). The graph will have one connected component when there exists at least one path linking any two vertices in the graph. However, the single connected component representing a cohesive class may have different levels of connectivity varying from: (i) the minimum component that could be formed by the set of vertices - the number of edges would be $|v| - 1$ - reflecting the minimum cohesion for a single component in the graph; (ii) to the maximum connectivity having all the methods (vertices) directly linked by an edge, and the number of edges is $|V| \cdot (|V| - 1)/2$ (maximum cohesion), forming a complete graph. The authors normalized the connectivity degree in the following expression:

$$Co(x) = 2 \cdot \frac{|E_x| - (|V_x| - 1)}{(|V_x| - 1) \cdot (|V_x| - 2)}$$

So, we always have $Co(x) \in [0, 1]$. Values 0 and 1 are taken for $|E_x| = |V_x| - 1$ and $E_x = |V_x| \cdot (|V_x| - 1)/2$, respectively.

²Originally defined as C, but used in Briand’s framework as *Co* to avoid name conflict.

This is an important refinement to distinguish different situations where the cohesion degree of a cohesive module varies. For classes with more than two methods, Co can be used as a discriminatory measure among the cases where LCOM4=1 but present different levels of cohesion.

Bieman and Kang (1995) also followed Chibamber and Kemerer, however they extend the notion of attribute usage. In their definition, two methods are called “connected”, if they directly or indirectly use a common attribute. A method m_1 indirectly use an attribute a if m_1 calls m_2 and m_2 uses attribute a . Given that observation, they proposed **TCC** (Tight Class Cohesion) and **LCC** (Loose Class Cohesion) metrics as follows:

Let $NDC(C)$ be the number of direct connections between public methods in a class C . Two methods are directly connected if there exists one or more common attribute between them.

Let $NP(C)$ be the maximum number of public method pairs. $NP(C) = [N \times (N - 1)]/2$, where $N = \text{number of public methods}$. Then TCC of a class C is the relative number of directly connected methods. $TCC(C) = NDC(C)/NP(C)$.

LCC is similar to TCC except that it considers the definition of indirect usage. Let $NIC(C)$ be the number of direct or indirect connections between public methods in a class C . $LCC(C) = NIC(C)/NP(C)$

For TCC and LCC, the authors identified a problem with constructor methods as they may indirectly connect any two methods which use at least one attribute. It artificially increases cohesion as measured by TCC and LCC. So they recommend to exclude constructors and destructors from the analysis of cohesion.

Henderson-Sellers (1996) formulated a new cohesion measure ranging from 0 to 1, called as **LCOM5**. Consider a set of methods $\{M_i\}$ ($i = 1, \dots, m$) accessing a set of attributes $\{A_j\}$ ($j = 1, \dots, a$). Let the number of methods which access each attribute be $\mu(A_j)$. Then he normalized the sum of $\mu(A_j)$ for all the attributes in a class in the following expression.

$$LCOM5 = \frac{\frac{1}{a} \sum_{j=1}^a \mu(A_j) - m}{1 - m}$$

This yields 0 when each method of the class references every class attribute, meaning the perfect cohesion. When the measure yields 1, each method of the class references only a single attribute. Values between 0 and 1 are to be interpreted as percentages of the perfect value.

Lee and Liang (1995) defined the Information-Flow-based Cohesion (**ICH**) metric. The idea behind ICH is that the cohesion of a method depends on the number of invocations to other methods and also the number of parameters used. The more parameters an invoked method has, the more information is passed, the stronger the link between the invoking and invoked method. The cohesion of a class is the sum of the cohesion of its methods. The cohesion of a set of classes is then the sum of the cohesion of the classes in the set. Thus, ICH can be applied to method, class and any set of classes.

Let $M_{OVR}(c)$ be the set of overridden methods in class c . It is applied if c is part of a class hierarchy, so being able to override inherited methods. And let $M_{NEW}(c)$ be the methods defined only in class c , not considering the overridden methods.

Consider $Par(m')$ be the set of parameters of method m' and $NPI(m, m')$ be number of polymorphic method invocations from m to m' . It is worth noting here that this definition considers not only static method calls but also all possible dynamic method calls which can be derived by object polymorphism.

$ICH^c(m)$ is then defined as the cohesion of method m in class c :

$$ICH^c(m) = \sum_{m' \in M_{NEW}(c) \cup M_{OVR}(c)} (1 + |Par(m')|) \cdot NPI(m, m')$$

While $ICH(c)$ is the cohesion of a class as a sum of of all the information-flow-based cohesion of its methods.

$$ICH(c) = \sum_{m \in M_I(c)} ICH^c(m)$$

Finally, Briand et al. (1998) defined the Ratio of Cohesive Interactions (**RCI**) metric. RCI considers different kinds of interactions among data (attributes and local variables) and methods when counting the number of connections between two methods or a method and an attribute.

There is a data-to-data interaction between attributes a and b if a change in a 's declaration or use may cause the need for a change in b 's declaration or use. For instance, if a public attribute a in a given class c is an array and its definition uses public constant b , there is a data-to-data interaction between b and a .

There is a data-to-method interaction between data declaration a and method m , if a interacts with at least one data declaration of m . Data declarations of methods include their parameters, return type and local variables. For instance, if a method m of class c takes a parameter of type class c , there is a data-to-method interaction between m and the implicit type declaration of class c . A thorough and more precise explanation about the different types of interactions can be found in (Briand; Daly; Wüst, 1998) and (Briand; Morasca; Basili, 1999).

RCI yields a value for the ratio between the existing interactions in a given class c (the $CI(c)$ set) and all possible interactions within that class (the $Max(c)$ set). The following expression represents this calculation:

$$RCI(c) = \frac{|CI(c)|}{|Max(c)|}$$

RCI=1 indicates that all the possible interactions within a class already exists (maximum cohesion), while RCI=0 means the minimum cohesion, i.e., no interactions among the class internal elements.

2.1.2 Metrics Comparison

All of the cohesion metrics presented in previous section are structural, which means that all of them calculate cohesion degree in terms of how the module internal elements are syntactically related to each other. However, it is worth noting that they present differences

in the manner in which cohesion is structurally addressed. This section summarizes the varying characteristics of the structural cohesion metrics which were previously discussed in literature (Briand; Daly; Wüst, 1998).

Domain of measure. The metrics may vary the domain of measure. Most of them were defined to measure the cohesion of a class, except the ICH measure which was explicitly defined to measure cohesion of a method, a class, or a set of classes.

Type of connection. Some of the metrics vary the type of connection considered to link elements (methods and attributes) within a class. For instance, LCOM1, LCOM2, LCOM3, LCOM4, TCC and LCC count pair of methods that use or do not use attributes in common. Besides that, LCOM4 and LCC also include method invocations as a mechanism to link methods within a class. Differently, instead of considering methods in pairs, other metrics capture the extent to which individual methods use attributes or locally defined types (LCOM5 and RCI), or invoke other methods (ICH). In addition and in contrast to the most of the metrics, ICH are focused only on method invocations. Class attributes are not considered at all.

Direct or indirect connections. Besides considering how the internal elements are directly related to each other, some metrics also consider indirectly connected elements. Such variation totally affects the counting mechanism for cohesion measurement. LCOM3, LCOM4, LCC and RCI are the ones that consider indirect connections.

Inheritance. In OO systems the use of inheritance is crucial for the design. However, most of the cohesion metrics do not discriminate whether to include inherited attributes and methods in the analysis. This lack of definition gives room for different interpretation and may drive distinct implementation by tool developers. Only for TCC, LCC and ICH, the authors explicitly address this situation. For TCC and LCC metrics, although the authors had not formalized the use of inheritance in their definitions, they discussed three alternatives for considering inheritance when applying their metrics: (i) include, or (ii) exclude inherited methods and attributes; or (iii) include inherited attributes but exclude inherited methods. For ICH metric, the authors exclude inherited methods in the analysis.

Accessor methods and constructors. This is another peculiarity of OO systems. It is common to have methods solely to provide read and write operations to a class attribute. Thus, a class may have several pairs of accessor methods which do not use any attributes in common. This constitutes a problem for metrics which count such pairs (i.e., LCOM1, LCOM2, and LCOM3). Another similar point is regarding the constructor and destructor methods. In this case is common to have classes with constructors accessing several attributes (e.g. attributes initialization), what directly influences the calculation of cohesion. Most of the metrics definitions do not give discriminatory treatment for those special kind of methods, also giving room for different interpretation and implementation. Only Co, TCC, ICH and RCI are free of problems regarding accessor methods and constructors due to the way they were defined.

2.2 CONCEPTUAL COHESION METRICS

There is a growing body of relevant work focusing on conceptual cohesion measurement (Etzkorn; Delugach, 2000; Marcus; Poshyvanyk, 2005; Liu et al., 2009; Ujhazi et al., 2010). They are called *conceptual* as the first metrics of this kind were described based on the term *concept*, which has the same meaning of *concern*. Here, we prefer to use the term *concern* instead of *concept*.

Most of the conceptual cohesion metrics rely on text mining-based techniques for automatically capturing what concerns each module addresses and then compute cohesion. We explain these approaches in next section. Also, in Section 2.2.2 we describe a recent conceptual cohesion metric proposed by our group which we have studied in this research scope. Then, in Section 2.2.3 we describe the concept of concern mapping which provides a fundamental mechanism for measuring conceptual cohesion.

2.2.1 Text Mining-based Cohesion Metrics

This section explains five conceptual cohesion metrics we found in literature that use text mining techniques for extracting information about concerns addressed by software modules.

The **Logical Relatedness of Methods (LORM)** (Etzkorn; Delugach, 2000) was the first attempt in this field. The counting mechanism for this metric relies on processing the source code text of each class for identifying concerns, and then mapping the identified concerns to a semantic network. For this computation, it is necessary to have in advance a semantic network which is formed by a graph of concerns. As the source code is processed links between source code elements and concerns of the semantic network are created. The resulting mapping is then used to calculate how the methods of a given class is semantically interconnected.

LORM is defined as:

LORM = the total number of (semantical) relations in the class / total number of possible (semantical) relations in the class.

A graph theoretically formalization can be found in (Etzkorn; Delugach, 2000). That notion of semantical interconnection differs from the structural interconnection as it is calculated based on concerns mined from the source code rather than evaluating the structural relations among the methods of a class.

Cox, Etzkorn and Hughes (2006) later extended the previous metric to calculate the semantic closeness among concerns by considering the indirect relations between pairs of concerns, as well as the direct relations. It is founded on the idea that the direct and indirect linkages between two concerns and other concerns in a domain-specific knowledge base reduce the ambiguity of those concerns. For example, consider a situation in which the terms in a GUI code class map onto a set of concerns like “toolbar”, “cursor” and “text size”. A knowledge base of GUI code concerns might not include direct relations between any of these three concerns. However, it would be quite likely that all three

concerns would be related to a fourth concern: “window”. This indirect relation shows that the three concerns are actually reasonably closely related, indicating that the class has good cohesion, which is a judgment that would be expected to be matched by human evaluators who would recognize all three concerns as relating to window control.

Therefore, they defined the metric **Semantic Closeness for Disambiguity (SCFD)**, which calculates conceptual cohesion considering indirect relations in a semantic network of concerns. Each link in a chain between two concerns generally serves to reduce the ambiguity of the concerns’ relationship. Two concerns in the knowledge base (assuming they are connected at all) are connected by a “chain” of one or more relations. The number of items in the chain is an indication of the level of definition of that concern within the knowledge base. The SCFD metric aggregates the distances between all the concerns from a given class. The following definition calculates the number of relations between all pairs of concerns and then the mean:

$$SCFD = \frac{1}{n(n-1)} \sum_{i=1}^n \sum_{j=1}^n L(i, j)$$

where n is the number of concerns and $L(i, j)$ is the number of relations in the shortest path between concern i and concern j .

The LORM and SCFD metrics lack empirical evidence showing the suitability of that approach. Also, their implementation are not available and the documentation on how to build the knowledge base of concerns is not sufficient. Therefore, it is difficult to apply those two metrics in recent empirical studies. However, they were the beginning of a series of related work on the search for improving cohesion measurement.

Marcus and Poshyvanyk (2005) proposed the application of Latent Semantic Indexing (LSI) (Deerwester et al., 1990). LSI works converting the source code under analysis into a text corpus, such that from each method only identifiers and comments are extracted. Each method of a class is considered as a document in this corpus and LSI is used to map each document to a vector in a multidimensional space determined by the terms that occur in the vocabulary of the software. A similarity measure between any two methods can be defined by calculating the similarity between their corresponding vectors. This similarity measure will express how much relevant semantic information is shared among the two methods, in the context of the entire system.

The computation of the similarity degree between methods of a class determines whether a class represents a single semantic abstraction, that is, whether a class is focused on a single concern. This is how the authors defined the **Conceptual Cohesion of Class (C3)** metric. C3 is the average conceptual similarity of the methods in a given class. Therefore, the more the methods of a class are written with comments and identifiers sharing similar terms the more conceptual related (i.e. cohesive) the class is. If the methods inside the class have low textual similarity, then the methods most likely participate in the implementation of different concerns and C3 will indicate low cohesion.

Later, Ujhazi et al. (2010) proposed a variation of the C3 metric called **Conceptual Lack of Cohesion of Methods (CLCOM5)**. It combines the counting mechanism of LCOM5, a graph-based structural cohesion metric explained in Section 2.1, with the

computation of the textual similarity of C3. As in LCOM5, CLCOM5 builds a graph linking the methods that have relations to each other. However, the graph relations between two methods are created if they hold textual similarity as defined in the C3 metric. Thereby, CLCOM5 of a given class is the number of connected components of a graph formed by the class methods as vertices and edges as conceptual relations between pairs of methods in the class.

Liu et al. (2009) proposed a conceptual cohesion metric called **Maximal Weighted Entropy (MWE)**, which uses the Latent Dirichlet Allocation (LDA) technique (Blei; Ng; Jordan, 2003). LDA is a more recent text mining approach than LSI, applied in C3 and CLCOM5. It is claimed to be better than LSI for many text corpora including software source code.

LDA is a statistical model, originally used in the area of natural language processing applied for text documents. The basic idea behind LDA is that text documents are represented as random mixtures over latent topics, where each topic is characterized by a distribution over words. For MWE computation, first it is necessary to run LDA assuming words as comments and identifiers declared in the source code of methods. Then methods are treated as documents, while classes are sets of documents comprising all the system which is the text corpus. As a result of LDA execution, a set of topics is identified, where each topic is a set of words. LDA associates topics to documents. Thus, the interpretation is that topics are concerns realized by the methods of each class in the system.

Once the distribution of topics (i.e. concerns) over the methods is computed, the next stage for MWE is to apply information theory to calculate the degree of which the methods of a given class is focused on a main topic. The class cohesion is measured by taking into account values of occupancy and distribution for the dominating topic, which is addressed in the class. If the maximum value for such a topic is low, this means that the class does not have a distinctive topic or a theme, which can be attributed to reduced class cohesion. Conversely, if the MWE value of a given class is high, this means that the class has a dominant topic, thus having high cohesion. MWE is formalized as:

$$MWE(C) = \max_{1 \leq i \leq |t|} (O(t_i) \times D(t_i))$$

Where $O(t_i)$ and $D(t_i)$ are respectively the occupancy and the distribution of topic i , having i representing each topic of the LDA execution. Further details on mathematical definition can be found in (Liu et al., 2009).

Authors of C3, CLCOM5 and MWE compared their metrics against other structural cohesion metrics with respect to fault-proneness. Their results pointed C3, CLCOM5 and MWE as complimentary fault-proneness indicators. However, they depend on reasonable naming conventions for identifiers and relevant comments contained in the source code. When these elements are missing, they suggest to rest only on structural metrics.

2.2.2 The Lack of Concern-based Cohesion Metric

In previous section we explained a set of conceptual cohesion metrics which use text mining-based techniques for computing cohesion. This current section defines the **Lack of Concern-based Cohesion (LCbC)** metric which was originally proposed by (Sant'Anna et al., 2007) with a different name. It is described here in a separate section as LCbC has different characteristics and does not depend on specific text-mining techniques as the other conceptual cohesion metrics in previous section.

The purpose of LCbC is to quantify cohesion of a given module in terms of the quantity of concerns addressed by it. A module can be a class, an interface, or whatever abstraction representing a module as a unit of implementation. Thus, it counts the number of concerns mapped to each module.

To unambiguously express the metric and facilitate the replication of our empirical studies, we present the formal definition of LCbC based on set theory. First, we present the terminology used on the formal definition. Let S be a system and $M(S)$ be the set of modules of S . Each module m consists of a set of attributes, denoted as $A(m)$, and a set of operations, represented as $O(m)$. The set of all attributes and all operations in system S are represented as $A(S)$ and $O(S)$, respectively. For each $m \in M(S)$, the set of concerns assigned to m is denoted by $Con(m)$. Let $o \in O(m)$ be an operation of m , and $Con(o)$ be the set of concerns assigned to o . Let $a \in A(m)$ be an attribute of m , and $Con(a)$ be the set of concerns assigned to a . For each concern con realized in the design of system S , the sets of modules, operations and attributes to which con is assigned are, respectively, denoted as:

$$\begin{aligned} M(con) &= \{m \mid m \in M(S) \wedge con \in Con(m)\}, \\ O(con) &= \{o \mid o \in O(S) \wedge con \in Con(o)\}, \text{ and} \\ A(con) &= \{a \mid a \in A(S) \wedge con \in Con(a)\}. \end{aligned}$$

The LCbC metric for a module m can now be defined as the number of elements in the set formed by the union of: the concerns assigned to the entire module m , the concerns assigned to each operation of m , and the concerns assigned to each attribute of m .

$$LCbC(m) = |Con(m) \cup \bigcup_{o \in O(m)} Con(o) \cup \bigcup_{a \in A(m)} Con(a)|$$

In Figure 2.1 we show a concrete example for LCbC measurement. It is the same example we present in Chapter 1. This figure illustrates a Java class with two attributes and nine methods, which is part of a system that builds and represents family relationships as trees. Particularly, this class represents the spouse relationship between two people: the husband and wife attributes. The methods involve: getting access to private attributes, getting the relation type, getting the partner object, objects comparison, and the relationship exporting to a given File. The light pink background highlights the class main concern, which is what the class is supposed to implement (i.e. representing spouse relationship). The blue background highlights the code fragments calling a window box to output a message in the user interface, and the green background highlights a method

to export the relationship represented by the class to a given file. Therefore, the value of LCbC for that class is three, as it addresses three concerns.

```

public class RelationSpouse extends Relation {
    public Person husband;
    public Person wife;

    public RelationSpouse (Person husband, Person wife){

    public String getRelationType(Person person){
        if (person.equals(husband))
            return WIFE;
        else if (person.equals(wife))
            return HUSBAND;
        else{
            JOptionPane.showMessageDialog(null,
                "Wrong person type for spouse relationship..."); output
            return INVALID_RELATIONSHIP;
        }
    }

    public Person getPartner(Person person){
        if (person.equals(husband))
            return wife;
        else if (person.equals(wife))
            return husband;
        else{
            JOptionPane.showMessageDialog(null,
                "Wrong person type for spouse relationship..."); output
            return null;
        }
    }

    public Person getHusband() {[]
    public Person getWife() {[]
    public Person getPerson1() {[]
    public Person getPerson2() {[]
    public boolean equals(Object obj) {[]
    public void printToFile(File f) throws IOException{[] Export to file
}

```

Figure 2.1 A sample class with mapped concerns

2.2.3 Software Concerns and Concern Mapping

Eaddy (2008) compiled a list of definitions for the term “concern” and its synonyms found in literature. In general, it has a broad definition as follows. A concern is generally interpreted as any property of interest for software project stakeholders. It is normally considered as a conceptual unit to be treated in a modular way (Tarr et al., 1999), (Robillard; Murphy, 2007), (Sant’Anna, 2008), (Figueiredo, 2009). Concerns can range from high-level properties such as quality of service and distributed communication to low-level properties such as data sorting and buffering. They can also represent functional properties like business rules and features, or non-functional such as transaction management and synchronization. Overall, typical concerns that have been reported in literature from real software systems are:

- functional and non-functional requirements from a requirements specification doc-

ument;

- use cases from a use cases specification document;
- roles from architectural or design patterns;
- implementation mechanisms (e.g. caching, buffering); and
- features from software product lines.

Concerns can be defined in the light of requirements engineering as well as emerge from implementation-level issues. Regardless what motivates the existence of concerns, conceptual cohesion measurement is focused on concerns that can be expressed in the source-code level, that is, concerns that can be explicitly represented by one or more lines of source code in terms of statements, methods and attribute declarations. For instance, “persistence” is an example of a concern which is usually realized by several modules in a typical information system. Another example of concern that is usually found in many systems is “login”. The conceptual cohesion metrics applied in this current research does not rely on concerns that influence how the system is built but do not trace to any specific code fragment. In particular, we do not focus on concerns that are not observable when the system executes, such as maintainability or readability.

Generally, conceptual cohesion metrics are supposed to rely on a *concern mapping*. Concern mapping is the assignment of software concerns to source code elements (e.g. attributes, methods, classes). Therefore, it involves two domains related to each other through a mapping relationship. The source domain is a set of concerns and the target domain is a set of code elements, as depicted in Figure 2.2. The arrows illustrate the assignment of distinct concerns to the corresponding source code elements that realize it. Thus, before computing conceptual cohesion metrics, it is expected to have a concern mapping to provide information about what concerns are addressed by source code elements of system modules. In OO source code, each concern is assigned to a set of statements, methods, attributes or entire classes that implement them.

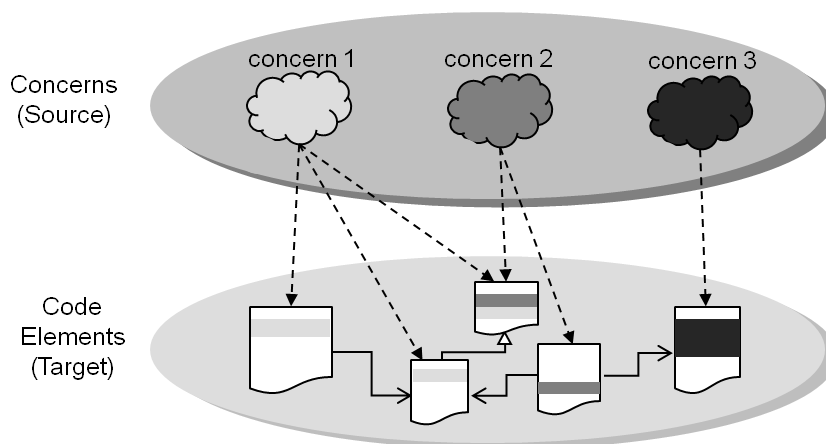


Figure 2.2 Illustrative concern mapping relationship

According to the example in Figure 2.1 the concern mapping source domain is formed by the three concerns identified in that class: spouse relationship object-oriented representation (light pink background), window-based message output as user interface mechanism (blue background), and file export (green background). The concern mapping target domain is formed by the code elements that are under the colored backgrounds corresponding to each concern identified.

The task of assigning concerns to source code fragments may vary depending on who are interested on the concern mapping and also on what artifacts are available. For example, from the requirements to code traceability perspective, a requirements specification document would be a good source of information to map concerns to source code (Eaddy et al., 2008b). From a product line engineering perspective, the feature model would be recommended as the source of information for concern mapping (Figueiredo et al., 2008). Also, when there is no high-level artifacts such as feature models or requirements specification documents, the developers knowledge can be used to assign concerns to code fragments (Eaddy et al., 2008b).

Moreover, there are some emergent technologies to automatically or semi-automatically identify concerns in source code, thus supporting the concern mapping task. They mainly vary on the level of automation (manual, semi-automatic or fully automatic). They may consider the kind of data, such as static information (e.g. source code text and structure), or dynamic information (e.g. execution traces). There are comprehensive reviews available in literature that address the different aspects of concern mapping techniques (Kellens; Mens; Tonella, 2007) (Dit et al., 2011). We summarize some of the approaches as follows.

Information retrieval techniques. Approaches that take the source code text as input and applies well-known information retrieval techniques for mining concerns that can be expressed by means of source code comments and identifiers (Marcus et al., 2004) (Poshyvanyk; Marcus, 2007), (Liu et al., 2009). Most of the approaches in this category are fully automatic. The main drawback of such information retrieval techniques is that mined source code text fragments may not be meaningful enough to represent software concerns.

Static analysis techniques. This category comprises approaches that process the source code syntactic structure to identify implemented concerns (Robillard; Murphy, 2007) (Trifu, 2009). They normally are semi-automatic techniques which need initial information as start point to find concerns. That initial information may be, for example, a method (called *seed*), manually defined by developers according to the concern they want to identify. After that, an automatic step is executed to identify the structural dependencies taking the seed method as input and then generating a set of methods and attributes as the suggested code elements to be assigned for the desired concern. The main drawback of this category is that the structural information undertaken may not be enough to represent some software concerns which are at higher abstraction levels. The syntactic structure may not be sufficiently expressive. Also, static analysis becomes too time-consuming when carried out semi-automatically.

Dynamic analysis techniques. This group relies on collecting information from a system during runtime (Eisenberg; Volder, 2005) (Bohnet; Voigt; Dollner, 2008) (Sartipi;

Safyallah, 2010). Most techniques in this category are semi-automatic. They usually require execution scenarios previously defined (e.g. test cases). As long as an execution scenario is executed, the execution traces are automatically collected and suggested as code fragments to be assigned as a concern implementation. This kind of technique may be too time-consuming for large systems where hundreds of executions scenarios may have to be executed to cover the distinct concerns the system implements. Moreover, execution scenario specifications are not always available.

History-based techniques. Approaches in this category rely on an evolutionary search for concerns in source code (Breu; Zimmermann, 2006) (Adams; Jiang; Hassan, 2010) (Hashimoto; Mori, 2012). They basically collect information from source code version control repositories and try to associate code fragments that had correlated evolution. These techniques strongly depend upon a rich change set to work as source of information. They are very sensitive to code changes. However, they may not be able to capture software concerns which are expressed in code fragments that rarely change or do not change at all.

Hybrid techniques. Several researchers have advocated hybrid approaches for concern mapping in order to take the benefits of different techniques and minimize the intrinsic limitations of individual approaches. For instance, Nguyen et al. (Nguyen et al., 2011) combined static analysis with clone detection techniques and code similarity algorithms. Antoniol and Gueheneuc (2006) put together static and dynamic analysis, while Eaddy et al. (2008a) applied in combination three distinct techniques - static analysis, information retrieval and dynamic analysis.

There is no consensus about the best approaches for concern mapping. The research community lacks benchmarks and empirical studies for comparing the list of techniques that have been proposed. In addition, it is still very difficult to find available tools, even in prototype versions, ready to be used either in real software projects or in research works. In the scope of our research, we used three different strategies for concern mapping: manual mappings made by developers; and two different tools – XScan (Nguyen et al., 2011) and TopicXP (Savage et al., 2010). XScan applies an automatic hybrid approach mixing their own information retrieval technique and static analysis, whereas TopicXP applies LDA, a well-known information retrieval technique. These tools and corresponding techniques consist of fully automatic mapping strategies for Java systems. As we studied only software projects written in Java, this is the main reason we used them. We are not aware of any other available fully automatic concern mapping tool for Java code. More details on how we used these tools are found in following chapters.

2.2.4 Summarizing Conceptual Cohesion Metrics

The first column in Table 2.1 lists all the conceptual cohesion metrics found in literature. This table also summarizes what we discuss in Section 2.2.

In terms of underlying mechanism for concern mapping, all of them except LCbC rely on information retrieval techniques such as text mining. Actually, LCbC is flexible by definition in the sense that it can be applied with any concern mapping technique, including text mining. It is different for example from C3 and CLCOM5 which use LSI by

definition, or MWE which relies on LDA. Conversely, LCbC has been used with different concern mapping strategies, as shown in next chapters. In addition, whereas technology on concern mapping improves its accuracy level over time, LCbC can take advantage over the other metrics by allowing the utilization of any new underlying concern mapping technique according to the purpose of who are using it.

Regarding the metrics implementation availability, only MWE and LCbC were made available. MWE is implemented in the original release of TopicXP tool, whereas LCbC was made available by us as we extended XScan and TopicXP to compute LCbC based on the underlying concern mapping techniques already implemented in those tools.

In terms of empirical assessment, LORM is the only metric that we did not find any application on empirical studies. The other ones were involved in empirical studies with mostly quantitative analysis. PCA (Principal Component Analysis), statistical correlation and regression analysis techniques were found in most of them. In addition to quantitative analysis, we also involved LCbC in a qualitative study, which is presented in Chapter 4.

Table 2.1 Conceptual cohesion metrics

Metric	Concern Mapping Approach	Availability	Empirical Assessment
LORM	Text mining with semantic network.	Implementation not available.	-
SCFD	Text mining with semantic network.	Implemented for C++ code, but not available for download.	Statistical correlation.
C3	Text mining with Latent Semantic Indexing (LSI).	Implemented for C++ code, but not available for download.	PCA, Correlation and Logistic regression to predict faults.
CLCOM5	Text mining with LSI combined with graph-based structural cohesion (LCOM5).	Implemented for C++ code, but not available for download.	Machine learning algorithms to predict faults.
MWE	Text mining with Latent Dirichlet Allocation (LDA).	TopicXP tool.	PCA and Logistic regression to predict faults.
LCbC	Any concern mapping technique.	XScan and TopicXP extensions implemented and made available by us.	PCA, Statistical correlation, Regression Trees, and qualitative assessment.

2.3 SUMMARY

This chapter summarized a set of conventional structural cohesion metrics which have been present in empirical studies along the years. Some of these metrics are also available in a number of tools for supporting software design and programming. The metrics original publications and also the Briand's framework (Briand; Daly; Wüst, 1998) served as the main source of information for the structural cohesion metrics presented in this

chapter. Also, we described all the five conceptual cohesion metrics available in literature so far. As a result, this chapter gives an overview about a representative set of cohesion metrics, which are important to understand their underlying interpretation and counting mechanisms for measuring cohesion. In addition, we applied some of the metrics explained here as part of the studies presented in chapters 3 to 6. In next chapter we present an empirical study showing evidence that conceptual cohesion metrics is orthogonal to structural cohesion ones.

UNVEILING A HIDDEN DIMENSION OF COHESION MEASUREMENT

Cohesion metrics may capture different *dimensions* of cohesion, meaning that they may be based on different information and counting mechanism and may reflect different interpretation of cohesion. In Chapter 2 we give an overview of several cohesion metrics divided in two categories: structural cohesion and conceptual cohesion. Those metrics have been target of most of the studies regarding module cohesion in the last decades.

The existence of conceptual cohesion metrics must firstly show scientific evidence that it measures cohesion in a different way most of the structural and conventional cohesion metrics already do. The conceptual nature of such alternative cohesion metrics is represented by other counting mechanisms than computing structural relatedness of module internal elements. Therefore, it is expected that conceptual cohesion metrics reflect a different interpretation of cohesion in comparison with the well-known structural cohesion metrics not only in terms of counting mechanism but also in terms of generated results. This is what we call as an additional dimension of cohesion measurement. This leads to the first research question presented in Chapter 1:

Q1. *Does the conceptual nature of conceptual cohesion metrics make them significantly different from structural cohesion metrics?*

We hypothesize that conceptual metrics significantly differ from structural metrics with respect to the information they use to quantify cohesion and, as a consequence, they unveil an additional dimension of cohesion measurement which is not captured by structural cohesion metrics. Therefore, the main goal of the study presented in this chapter is to provide evidence about whether this hypothesis is not rejected. We carried out an empirical assessment where we statistically analyzed whether or not conceptual metrics capture at least a new dimension of module cohesion that is not captured by a representative set of structural cohesion metrics. This work represents a first investigation focused on the role of conceptual cohesion metrics proposed by different authors in the context of industry-scale long-lived systems. As a consequence, this study could justify further investigations on understanding the performance of conceptual cohesion

in contrast to structural cohesion metrics. Section 3.1 explains study settings, and Section 3.2 presents and discuss the results. In Section 3.3 the study threats to validity are then described. Section 3.4 summarizes related work, and finally Section 3.5 presents the study conclusion.

3.1 STUDY SETTINGS

The study design is presented in the following sections. First, we explain the selection of six open-source systems which were analyzed in this study. Second, we present how we measured conceptual cohesion. Third, we explain the set of five structural cohesion metrics we used and how we applied them. Finally, in the last two sections we discuss the method of analysis and the study results, respectively.

3.1.1 Selected Systems

We selected six open-source software systems with different sizes and from distinct domains: chart library, language interpreter, text editor, web container, program analysis tool and game. Table 3.1 summarizes the systems under study. Together they have 581,308 lines of source code (excluding comments and blank lines) and 3,733 Java files. All of them are long-lived projects. Most exist for more than 10 years, such as Tomcat, Rhino, JFreeChart, Freecol and jEdit. Also, these systems have been in widespread use. For example, Tomcat is one of the most traditional and reliable Java web containers available. Rhino is the default Java scripting engine embedded in the official Oracle’s Java Development kit. JFreeChart is the most widely used chart library for Java with millions downloads from the repository. Findbugs has been used by many major companies such as Google. In addition, we intentionally selected these systems from different domains to avoid domain-related bias.

Table 3.1 Summary of the systems under study

	JFreeChart	Rhino	jEdit	Tomcat	Findbugs	Freecol
Release	1.0.6 (June/2007)	1.6R5 (Nov/2006)	4.3.2 (May/2010)	6.0.26 (Mar/2010)	1.3.5 (Sep/2008)	0.84 (Aug/2009)
Description	Chart library for the Java platform	Mozilla’s Javascript interpreter	Text editor for programmers	web server container for the Java platform	Static analysis tool to find bugs in programs	A civilization-like game
LOC	76,059	59,182	109,516	161,735	98,914	75,902
# of Java files	514	156	531	1060	1041	431

Table 3.1 also indicates the selected releases of each system. They represent stable versions of each system which had been downloaded and used world-wide.

3.1.2 Conceptual Cohesion Measurement Procedure

As presented in Chapter 2, there are six conceptual cohesion metrics available in literature, according to our research: LORM, SCFD, C3, CLCOM5, MWE and LCbC. SCFD and CLCOM5 were proposed as evolution of LORM and C3, respectively. However, LORM, SCFD, C3 and CLCOM5 do not have available implementations for download. Therefore, we use LCbC and MWE as the representative set of conceptual cohesion metrics for this study. These metrics rely on different counting mechanisms than computing structural relation among internal class members.

3.1.2.1 Measuring LCbC using XScan LCbC depends on a concern mapping, which is a time-consuming task when performed manually. The selected systems under study are large enough to become unfeasible the manual concern mapping for all of them. Therefore we chose a concern mapping tool called XScan (Nguyen et al., 2011) in order to perform the study in a reasonable time. XScan automatically generates the concern mapping by collecting groups of methods that participate together in the realization of a concern. First, XScan searches for pairs of methods as candidates to be part of a concern. Two methods will be part of a concern if they satisfy at least one of the following conditions: (i) they have similar portions of code in their body; (ii) they override or implement the same ancestor method; or (iii) they have similar names. Then, after all possible pair of methods are detected, XScan builds a graph in which nodes represent the methods and edges represent methods relations. Each connected component of that graph might be reported as a concern. Actually, this is an approach that puts together different techniques which can heuristically suggest groups of methods as concerns realizations mined from the source code.

This tool has been proposed recently and the authors reported results indicating more than 90% of accuracy for finding concerns which are spread over several files (Nguyen et al., 2011). The complete XScan output with the concern mappings for our study is available at the companion website (Silva, 2015a). This output shows a list of method groups each one realizing a concern.

Table 3.2 shows a summary of XScan applied over the systems under study. Tomcat, which is the biggest system we analyzed, had the highest number of concerns identified by XScan, while Freecol had the highest file coverage, i.e. 56% of Java files with at least one concern mapped. For this study and hereafter in this dissertation, we consider a module as a Java file representing a class, an interface or whatever abstraction representing a module as a unit of implementation. For Java files with nested classes the predominant class was defined as the module and the concerns of its nested classes were summed up to concerns of the predominant class.

Originally XScan does not computes any metric. Thus, we enhanced XScan to compute LCbC over the concerns mapped by the tool and then to output this computation as a csv¹ file. Our extended version of XScan is available at (Silva, 2015a). The last four columns 3.2 show information about LCbC distributions of all systems under study using

¹Comma separated values

Table 3.2 Summary of XScan applied and corresponding LCbC distributions

Systems	# of concerns mapped	% of java files with concerns	LCbC_XScan			
			Min	Max	Median	Std Dev.
JFreeChart	208	46%	0	37	0	3.5
Rhino	26	32%	0	11	0	2.46
jEdit	99	36%	0	9	0	1.35
Tomcat	315	35%	0	20	0	2.48
Findbugs	151	32%	0	10	0	1.62
Freecol	91	56%	0	13	1	2.13

XScan. We hereafter call LCbC_XScan for explicitly referring to LCbC computed using XScan as the underlying concern mapping strategy.

3.1.2.2 Measuring MWE using TopicXP For measuring MWE it is necessary to execute LDA (Latent Dirichlet Allocation) (Blei; Ng; Jordan, 2003) topic modeling and then compute the entropy of identified topics in each class. LDA is a generative probabilistic model for collections of discrete data such as text corpora. LDA is used to automatically discover a set of topics within a corpus. A corpus is formed by a set of documents. Each document is expressed as a probability distribution of topics. Each topic is itself a probability distribution of words that co-occur frequently in a corpus of text. Words can belong to multiple topics, while documents can contain multiple topics.

LDA has been used as a machine learning technique in a growing number of software engineering problems (Thomas et al., 2010) (Grant; Cordy; Skillicorn, 2012) (Binkley et al., 2014) including the identification of concerns in source code (Linstead et al., 2007) (Baldi et al., 2008) (Maskeri; Sarkar; Heafield, 2008). Due to the nature of language usage, the words that constitute a topic are often semantically related (Thomas et al., 2010). An example topic is “patient clinic therapy medicine diagnostic”, which describes the healthcare industry. Therefore, there is a strong conceptual similarity between latent topics and the concept of software concerns (Baldi et al., 2008).

To apply LDA for source code analysis, topic extraction and MWE computation we used TopicXP (Savage et al., 2010). TopicXP calculates MWE by taking a Java project as the document corpus, methods as documents, and source code comments and identifiers as words. This tool automatically pre-processes documents for splitting words, removing stop words, and word stemming. In addition, we extended TopicXP to remove license comments typically present at the header of each open source java file in order to avoid noise in the document corpus. Our extended version of TopicXP can be found at our companion web site (Silva, 2015a).

Table 3.3 shows the LDA parameters we set up for this study. First, LDA requires the desired number of topics as input to run the algorithm. Then, we applied the following equation to find the number of topics t given a number of documents d (Java files):

$$t(d) = 7.25 * d^{0.365}$$

This equation was proposed by (Grant; Cordy; Skillicorn, 2012) (Grant; Cordy, 2010) after empirical assessment of LDA application over source code analysis. The optimal number of topics to build the LDA model for a source code corpus is an open question in source code analysis. Grant and colleagues' work is the first initiative to determine the appropriate number of topics needed to accurately represent source code in LDA models. Defining the proper number of topics is not a trivial task and requires a reasonable effort, thus becoming itself an open research topic for specific investigation. Therefore, it is out of our research scope investigate a method to find the best number of topics for our target systems. These are the main reasons we decided to follow Grant's findings and proposed equation.

α and β (third and fourth columns respectively in table 3.3) are also important parameters for LDA execution. According to (Binkley et al., 2014) a small value of β favors fewer words per topic. When α is small the LDA model emphasizes fewer topics per document. Conversely, when α increases the LDA model moves toward encouraging more topics per document. So the combination of small β and high α enables (but does not require) documents associated to more topics which are simpler topics with a smaller number of dominant words. We followed this reasoning because, in theory, this may lead to more comprehensive topics while supporting the increase of document-topic probability.

Therefore, the β parameter is at low level (0.1) for all the systems, which is also the default β of TopicXP tool. For α , TopicXP has the default value as $50/K$ (where K is number of topics). However, in order to combine low β with high α , we decided to increase α to $100/K$.

The last parameter is the threshold value, which is used to filter out the topics which are not associated to a document, that is, topics with a low probability to be in the document. We defined this threshold as 0.1 for all the systems. Therefore, we parameterized TopicXP to consider a topic within a document if the topic has a probability greater than 0.1 to appear in the document. This low threshold also favors more topics associated to a document.

Table 3.3 also shows some descriptive statistics information of MWE measurement (min, max and median values and standard deviation). MWE varies from 0 to 1 and different from other metrics in this study the higher the MWE of a class the higher is the class cohesion.

Table 3.3 LDA parameters applied and a summary of MWE distributions

Systems	# Topics	α	β	Threshold	MWE			
					Min	Max	Median	Std Dev.
JFreeChart	70	1.4	0.1	0.1	0	0.19	0.01	0.35
Rhino	40	2.5	0.1	0.1	0	0.4	0.02	0.07
jEdit	69	1.4	0.1	0.1	0	0.19	0.01	0.21
Tomcat	92	1.08	0.1	0.1	0	0.27	0.01	0.43
Findbugs	91	1.09	0.1	0.1	0	0.3	0.01	0.35
Freecol	66	1.5	0.1	0.1	0	0.69	0.01	0.1

The MWE measurement can be reproduced by executing TopicXP with the same parameters we describe in this section, whereas LCbC can be reproduced by executing XScan over the analyzed systems without any given parameter.

3.1.3 Structural Cohesion Measurement

We chose a set of five structural cohesion metrics. They represent four variations of the well-known Lack of Cohesion in Methods (LCOM): (i) LCOM2 (Chidamber; Kemerer, 1994), which is the adjusted version of LCOM1 proposed by Chidamber and Kemerer in 1991; (ii) LCOM3 (Hitz; Montazeri, 1995), a graph-based version of LCOM1; (iii) LCOM4 (Hitz; Montazeri, 1995), which is an expanded version of LCOM3; and (iv) LCOM5 defined by Henderson-Sellers (1996) in his book of software metrics. Besides those versions of LCOM, we also included the Tight Class Cohesion (TCC) proposed by Bieman and Kang (1995). We recognize that there are other variations of structural cohesion metrics available in the literature as described in Chapter 2. However, we believe that this set of five metrics is representative of available measures for structural cohesion since: (i) they were object of several studies in the last years (Briand; Daly; Wüst, 1998), (Marcus; Poshyvanyk; Ferenc, 2008), (Etzkorn et al., 2004), (Dallal; Briand, 2012); (ii) each metric reflects distinct choices for calculating structural cohesion; (iii) such choices were originated from a group of authors who have conducted studies about well-known cohesion metrics in the lastest years; and (iv) these metrics are supported by several tools.

The last step was to apply the five structural cohesion metrics on the modules of the selected release of each system. The Together environment (Together, 2011) was used to compute LCOM2, LCOM3, LCOM5 and TCC, and the Analizo metrics tool (Terceiro et al., 2010) was used to compute LCOM4. Two different tools were used because no available single tool supports all the five metrics for Java systems. Measurement results for all systems are available at the companion website (Silva, 2015a).

In summary, we gathered values for seven metrics: five from structural cohesion metrics, and two from conceptual cohesion. All these metrics were computed per module. Given the 3,733 Java files, this totaled about 22,000 data points. We developed a program to merge all the metrics computation output of each system in one single file to be used as input to the R statistical tool. All the cohesion measurement results are available at (Silva, 2015a).

3.1.4 Method of Analysis

The study research question is concerned with investigating whether conceptual cohesion metrics (LCbC and MWE) captures at least an additional dimension of cohesion measurement in comparison with existing structural cohesion metrics. Targeting the research question, we applied Principal Component Analysis (PCA) (Jolliffe, 2002) in order to help us understand the different dimensions captured by the cohesion measures. The application of PCA was previously suggested by Briand et al. (2000) in his methodology for analyzing software engineering data through the use of metrics and it has been applied in several other studies involving cohesion measurement (Chae; Kwon; Bae, 2000)

(Marcus; Poshyvanyk; Ferenc, 2008) (Liu et al., 2009). The overall purpose of PCA is to identify groups of variables — in our case groups of cohesion metrics — that explain as much of the variation with as few variables as possible. Each group of variable, called as principal component, is likely to measure the same underlying dimension of the object to be measured. In our case, the object to be measured is the cohesion of a class. Anyone can reproduce our PCA assessment by executing the R scripts we made available in Appendix A and in our companion website (Silva, 2015a).

3.2 STUDY RESULTS AND DISCUSSION

Figure 3.1 shows the principal components (PCs) that resulted from executing PCA on the cohesion measurements for each system. Those resulting components were generated by using the R tool and the varimax rotation technique (Jolliffe, 2002). Results in Figure 3.1 are grouped by system, having each group including five principal components. Those five components encompass weighted combinations of metrics which account for as much sample variance as possible. For example, in JFreeChart, LCOM3, LCOM5 and TCC are the main contributors of the first principal component (PC1) variance. Their coefficients are highlighted in the corresponding column. This means that LCOM3, LCOM5 and TCC define one dimension of cohesion measurement for this system, and this dimension accounts for 33% of sample variance. For each component, we provide the corresponding proportion (third row) in terms of the variance of the data set, which is explained by that component, and also the cumulative variance (fourth row). Following the JFreeChart example, besides the PC1 dimension, PC2 is formed solely by LCbC_XScan, which accounts for 14% of variance. The last column (PC5) of each system shows the cumulative variance which is the sum of proportional variance of principal components. Overall, each column of Table 3.1 represents a principal component which may explain a dimension of cohesion measurement. Therefore, when looking at a column, each highlighted cell in that column represents the main metrics that contribute for the variance of the corresponding component.

3.2.1 Unveiling Conceptual Cohesion Dimensions

The study research question explores whether or not conceptual cohesion metrics capture a dimension of module cohesion that is not captured by structural cohesion metrics. LCbC_XScan row in Figure 3.1 shows that this metric was the major metric of a principal component in all systems. Also, in all systems LCbC_XScan contributed almost exclusively for a principal component. In JFreeChart, Tomcat and Findbugs, LCbC was the main metric of PC2; in Rhino and jEdit, LCbC was the main metric responsible for PC5; and in Freecol it was the main metric of PC4.

Turning to MWE, results also indicates that it defines an additional dimension of cohesion measurement. The last row shows that MWE was the major metric of: PC1 in Freecol, together with LCOM4; PC3 in JFreeChart, Tomcat and Findbugs; PC4 in Rhino and jEdit.

Besides executing PCA for five principal components, we also executed with different

	JFreeChart					Rhino					jEdit				
	PC1	PC2	PC3	PC4	PC5	PC1	PC2	PC3	PC4	PC5	PC1	PC2	PC3	PC4	PC5
Proportion	0.33	0.14	0.14	0.14	0.14	0.27	0.21	0.18	0.14	0.14	0.32	0.15	0.14	0.14	0.14
Cumulative	0.33	0.48	0.62	0.77	0.91	0.27	0.48	0.67	0.81	0.95	0.32	0.47	0.61	0.75	0.90
LCOM2	0.10	0.08	0.03	0.25	0.96	0.95	0.04	0.12	0.11	-0.05	0.07	0.98	0.09	-0.02	0.03
LCOM3	0.89	0.15	0.08	0.12	0.08	0.21	0.71	0.57	0.07	0.04	0.90	0.12	0.15	0.06	0.08
LCOM5	0.88	0.03	0.04	-0.06	0.12	0.13	0.16	0.96	0.08	0.03	0.86	0.13	-0.04	0.13	0.08
TCC	-0.85	-0.01	0.05	-0.15	0.03	-0.10	-0.97	-0.08	0.01	-0.05	-0.79	0.14	-0.22	0.07	0.04
LCOM4	0.12	0.09	0.03	0.95	0.25	0.94	0.19	0.10	0.03	-0.05	0.19	0.10	0.96	0.05	0.08
LCbC_XScan	0.10	0.99	0.02	0.08	0.07	-0.07	0.06	0.04	0.01	1.00	0.07	0.03	0.07	-0.03	0.99
MWE	0.03	0.02	1.00	0.02	0.03	0.10	0.02	0.08	0.99	0.01	0.07	-0.02	0.05	0.99	-0.03

	Tomcat					Findbugs					Freecol				
	PC1	PC2	PC3	PC4	PC5	PC1	PC2	PC3	PC4	PC5	PC1	PC2	PC3	PC4	PC5
Proportion	0.33	0.14	0.14	0.14	0.14	0.32	0.14	0.14	0.14	0.14	0.25	0.21	0.16	0.16	0.14
Cumulative	0.33	0.48	0.62	0.77	0.91	0.32	0.47	0.61	0.76	0.90	0.25	0.46	0.63	0.78	0.92
LCOM2	0.10	0.08	0.03	0.25	0.96	0.13	0.04	0.01	0.14	0.98	0.17	0.13	-0.08	0.22	0.95
LCOM3	0.89	0.15	0.08	0.12	0.08	0.90	0.11	0.08	0.10	0.12	0.12	0.71	-0.46	0.29	0.04
LCOM5	0.88	0.03	0.04	-0.06	0.12	0.88	0.04	0.05	-0.02	0.06	0.04	0.93	-0.10	-0.13	0.11
TCC	-0.85	-0.01	0.05	-0.15	0.03	-0.81	0.05	0.04	-0.13	-0.03	-0.07	-0.22	0.95	-0.03	-0.08
LCOM4	0.12	0.09	0.03	0.95	0.25	0.12	0.00	0.01	0.98	0.14	0.92	0.10	-0.13	0.17	0.10
LCbC_XScan	0.10	0.99	0.02	0.08	0.07	0.06	0.99	0.09	0.00	0.04	-0.01	-0.01	-0.05	0.95	0.21
MWE	0.03	0.02	1.00	0.02	0.03	0.05	0.09	0.99	0.01	0.01	0.94	0.03	0.01	-0.16	0.10

Figure 3.1 Principal components of each system

number of components (3, 4 and 6). We found five components the best choice as they could account in all systems with 90% or more of sample variance (see Cumulative row for each PC5). This is one of the suggested criterion to define the number of components parameter when executing PCA (Jolliffe, 2002).

In summary, both conceptual cohesion metrics defined a principal component in this analysis. LCbC_XScan was the only major metric of a component in all systems. MWE performed almost in the same way, except in Freecol that MWE mainly formed PC1 with a structural cohesion metric (LCOM4).

It is understandable that LCbC_XScan and MWE define different dimensions of cohesion measurement. In fact, they use distinct counting mechanism for computing cohesion. In this study, we used XScan as a tool for generating the underlying concern mapping and then computing LCbC. As explained in Section 3.1.2.1, XScan analyzes method identifiers combined with their calling context in order to identify groups of correlated methods, where each group we considered as a concern. For MWE, there is the LDA topic modeling technique as an underlying text mining mechanism to identify topics, interpreted as concerns, through processing source code comments and identifiers. MWE uses LDA topic modeling output to compute topics distribution and entropy within classes as a

measure of cohesion.

Therefore, we can conclude that conceptual cohesion defines an additional dimension of class cohesion, at least when represented by LCbC_XScan or MWE. Such additional dimension demonstrated by PCA is explained by: (i) the underlying concern mapping mechanism used by a conceptual cohesion metric; (ii) how a metric uses concern mapping results to compute cohesion; and (iii) by generating cohesion distributions with different variance. The study conclusion scientifically supports our hypothesis that the underlying mechanism of conceptual cohesion is different from the structural metrics by reflecting different information about module cohesion.

Besides the quantitative analysis of PCA, we also discuss some examples that illustrate the differences on the dimensions of cohesion captured by LCbC, MWE and structural cohesion metrics. We found some modules in different systems that match the following pattern: low conceptual cohesion and high structural cohesion. For instance, in Tomcat the *StandardManager* class, which is used for managing the pool of sessions in the web container, had 8 concerns assigned to it. It is the 31st in the LCbC rank out of 1060 Java files analyzed in Tomcat, which means this is in the top 3% least cohesive classes according to LCbC. We analyzed the class source code and found that the 8 concerns are related to: start/stop operations for session loading/unloading; session creation; and the roles of the Observer design pattern. On the other hand, the LCOM2 value for this class was 158, which can be considered a low value as it represents the 145th in LCOM2 rank, placing in the top 13% least cohesive classes according to LCOM2. Therefore, in such case the conceptual dimension, measured by LCbC, captured better the information about cohesion as we observed that the class indeed has low cohesion as it addresses several concerns of the software.

Another example is the *DefaultPlotEditor* class in the JFreeChart system. According to MWE it is in 13% least cohesive classes. However, its LCOM4 value is 1, placing it as in 64% least cohesive classes. This low LCOM4 value does not reflect the number of distinct concerns in the class, such as drawing, zooming, axis space, click handling and plotting.

There is a similar example in the Freecol game system. The class *ReportColonyPanel* is a panel for displaying the colony report. According to LCbC, it is in top 10% least cohesive classes (LCbC_XScan = 4), whereas according to LCOM2, it is in top 31% least cohesive classes (LCOM2 = 15). This is another example where a conceptual cohesion metric reflected a dimension of cohesion different to the one reflected by a structural metric.

However, the opposite was also observed. As conceptual cohesion strongly depends on the underlying mechanism to find concerns associated to the modules of a project, it may fail whenever such task generates incomplete results by not succeeding on identifying some important concerns the modules realize. Regarding LCbC_XScan, we found some modules without concerns mapped to them (LCbC = 0) while they actually realized at least one important concern. We observed similar cases with MWE. These are modules whose measurement matched the following pattern: high conceptual cohesion, low structural cohesion.

In the jEdit system, we found that the *jEdit* class has the 3rd highest value in the

LCOM2 rank. Although we observed several concerns realized by this large and central class, which has 4,373 lines of code, the LCbC value for it was zero. In fact, XScan was not able to assign concerns to any of the members of this class. Turning to MWE, jEdit class stayed in the median of MWE distribution for this system. We analyzed the topics identified for such class by LDA execution and found that they did not reflect well the responsibilities addressed by jEdit class.

In the same system, we can also highlight the *JEditBuffer* class, which represents the contents of an open text file as it is maintained in the computer's memory. This class has the 4th highest LCOM2 value. However LCbC was zero for it. Again, this is an important class for a text editor and should have had concerns mapped to it, at least one concern about text editing. This is another case where the concern mapping strategy used by a conceptual cohesion metric did not perform well and, consequently, hinder the performance of conceptual cohesion when representing the actual responsibilities a module addresses.

Switching to the Findbugs system, there are situations worth mentioning as well - the *SortedBugCollection* and *OpcodeStack* classes. Both classes have no concerns mapped to them. However, they were on the top 10 highest values for LCOM2.

Besides statistically finding an additional dimension of cohesion measurement, these results also points out the need to investigate the use of different concern mapping strategies in order to understand their impact on conceptual cohesion measurement. We address this issue in Chapter 6.

3.3 THREATS TO VALIDITY

We identified threats related to internal and external validity which are discussed in this section.

Internal validity. The quality of the concern mapping is certainly an issue that affected conceptual cohesion measurement. For LCbC, we used XScan for producing the concern mapping. As we discussed in previous section, XScan does not detect some concerns in the system, because they are not expressed through methods with similar names or similar code and also do not participate in the same calling context. However, we decided to use this tool because it provides a fully automated step for mapping concerns, and their authors reported more than 90% of accuracy on identifying crosscutting concerns (Nguyen et al., 2011) in some systems they analyzed. We did not find in the literature an available tool ready to be used in real projects with better accuracy which could generate concern mappings fully automatic for Java systems.

Another threat regarding the XScan tool is that part of its heuristics relies on structural information in source code. It checks whether methods override or implement the same ancestor method. To some extent, the use of structural information for generating a concern mapping should be avoided, as we claim that LCbC is a conceptual cohesion metric. However, XScan heuristics are also based on more complex information, such as similarity of method names and detection of similar portions of code. In Method names and similar portions of code may represent semantic information reflecting the implementation of a concern or part of it.

For MWE, we used TopicXP tool for producing the LDA topic distribution. Two possible threats in this case are: whether TopicXP implements well the LDA technique; and whether we had the best choices for LDA parameters. We used TopicXP mainly because in a single tool it computes LDA algorithm and also calculates MWE. In order to improve TopicXP, we extended the tool to eliminate license comments within classes to provide textual noisy reduction. Regarding the chosen LDA parameters, it is empirical and still an open research question. To mitigate this threat we followed recommendations from recent studies focusing on LDA execution for software engineering artifacts such as source code (Grant; Cordy; Skillicorn, 2012; Binkley et al., 2014).

External validity. Our analysis focused on five structural and two conceptual cohesion metrics, involving six systems. The small sample size in terms of number of systems and metrics may be a threat to external validity. To minimize this, we selected industry-scale, long-lived, medium to large-sized systems from distinct application domains such as game, chart library, text editor, web container and programming language interpreter. In terms of programming language, all the systems are written in Java. However, it might not be a threat to external validity as the metrics can also be applied to other object-oriented languages and our analysis did not observe any specific Java characteristic.

Regarding the chosen metrics, the set of five structural cohesion metrics are well-known metrics in literature proposed by different authors and available in programming environments and tools. The set of available conceptual cohesion metrics in literature is small. According to our research, besides LCbC, there are five conceptual cohesion metrics proposed in literature (see Chapter 2). They are text mining-based cohesion metrics, from which only MWE has an implementation available for download. Also, MWE uses a more recent text-mining technique (LDA). Therefore, we did not find worth implementing C3 or CLCOM5 for our study settings. Ultimately, LCbC has been studied in our research group for years and it is flexible to be applied with any concern mapping strategy. Therefore, the conceptual cohesion metrics we used, MWE and LCbC, are the best set we could have for representing conceptual cohesion metrics. Also, we do not claim that our conclusions can be generalized outside this scope. However, with this scope of analyzed systems and metrics, we could show enough evidence to support us answering the research questions and giving us clues to further investigations.

3.4 RELATED WORK

In Chapter 2 we presented all the conceptual cohesion metrics we identified in the literature in the course of our research activities. Some of them also applied PCA in a similar way we did in the scope of this study.

Marcus, Poshyvanyk and Ferenc 2008 applied PCA in order to understand the underlying orthogonal dimensions captured by C3 and other structural cohesion metrics. The difference to our approach is that we did not have the goal of analyzing a specific conceptual cohesion metric contrasting with structural cohesion metrics, like they did with C3. We had a broader scope as we applied PCA to provide evidence that conceptual cohesion metrics measure cohesion in a different way. Such different way is explained by a different counting mechanism and source information focusing on computing cohesion

by capturing concerns each class addresses. Also, for that purpose, we used two conceptual cohesion metrics which rely on distinct concern mapping strategies (refer to Section 3.1.2 for a complete explanation about the concern mapping strategies used for LCbC and MWE).

Liu et al. (2009) also applied PCA to investigate whether MWE captures aspects of class cohesion that are not captured by other cohesion metrics. In this study they went a little further compared to C3 study, as they used two conceptual cohesion metrics, MWE and C3, besides conventional structural cohesion representatives. However, although C3 and MWE are different conceptual cohesion metrics, they both use text mining techniques for retrieving information about concerns each class addresses. In our study, the conceptual cohesion metrics used distinct concern mapping techniques: LCbC used XScan strategy, whereas MWE used LDA topic modeling.

Additionally, PCA in C3 study involved three systems, whereas in MWE work it involved only one system. In our study we applied PCA for six systems which were presented in Section 3.1.1.

In terms of results, the findings from the quantitative assessment of PCA are similar among all the three studies. However, in addition to the quantitative assessment, we also discussed with examples some cases that illustrate the differences on the dimensions of cohesion captured by LCbC, MWE and structural cohesion metrics.

3.5 SUMMARY

This study is a first-step for providing empirical evidence about the uniqueness of conceptual cohesion when compared to conventional structural cohesion metrics. Our initial assumptions pointed to the fact that the most well-known cohesion metrics fail to capture the actual notion of module cohesion as it is defined. This has motivated our investigation about whether the conceptual nature of conceptual cohesion metrics really makes them different from structural cohesion metrics not only in terms of counting mechanism but also in terms of generated results.

In this context, we carried out an empirical assessment with the support of statistical methods involving six open-source, industry-scale, long-lived, medium to large-sized systems. We found that conceptual cohesion is orthogonal to structural cohesion, meaning that a different dimension of cohesion measurement is captured by conceptual cohesion metrics due to different source of information and counting mechanism for computing cohesion. Additionally, we highlighted and discussed typical situations where the conceptual cohesion metrics worked well or not for quantifying cohesion. This gives us insight for further investigation presented in Chapter 6.

The overall standpoint is that our findings showed that conceptual cohesion metrics are promising and worth investigating further as an alternative way of cohesion measurement. Besides evidence that conceptual cohesion metrics measure cohesion differently, we do not know whether such difference reflects how developers reason about module cohesion. This issue is explored in the next chapter. In theory, when decomposing a problem into modules it is easier to reason on what concerns each module should address. In other words, it is supposed to be more natural for developers to achieve high cohesion when

it is perceived by using abstract information such as concerns like conceptual cohesion metrics do. This thinking also applies when developers have to maintain existing modules by addressing new concerns or changing existing ones.

HOW DEVELOPERS REASON ABOUT MODULE COHESION

According to Etzkorn et al. (2000), “the cohesion metric that would be preferred by software engineers would be a metric that best reflected a human-oriented view of cohesion”. For them, highly cohesive modules should have logically related code no matter if they are syntactically related or not. This raises the abstraction level for measuring cohesion as it depends on what each module implements rather than on how interrelated are the internal elements of each module. This view of cohesion is also shared by other authors (Marcus; Poshyvanyk, 2005), (Marcus; Poshyvanyk; Ferenc, 2008), (Liu et al., 2009).

In the last chapter we showed empirical evidence that conceptual cohesion represents an additional dimension of module cohesion when compared to other conventional structural cohesion metrics. However, it is still unaddressed to what extent such structural and conceptual cohesion metrics reflect developer’s perception of module cohesion. Presumably, it is easier for humans, when decomposing a problem into modules, to find distinct concerns implemented by a module than to mentally calculate the degree of structural relatedness amongst methods and attributes. However, there is no empirical evidence to support this argument. Moreover, there is no evidence whether or not developers know the concept of module cohesion as it is in software engineering theory. Therefore, the main goal of this study is to provide empirical evidence about how developers perceive module cohesion and assess to what extent such perception associates with structural and conceptual cohesion measurement.

To achieve this goal, we performed an empirical study where we investigated: (i) what rationale developers used to rate cohesion of different modules and (ii) to what extent the ratings they gave were related to structural and conceptual cohesion measurements. The study included a web-based closed-access survey involving 80 participants from nine countries and different levels of experience and academic degrees. The survey comprised questions related to: general perception of module cohesion; module cohesion comparison and rating; cohesion reasoning; and participant profile.

The contribution of this study is threefold. First, we built up empirical evidence about how developers perceive cohesion and which cohesion measurement associates with their opinion. It represents a stepping stone towards understanding the applicability of structural and conceptual cohesion measurement. Second, we found that, in the context of our study, conceptual cohesion metrics were better representative of developers' perception of cohesion than structural metrics. Finally, the study design, the survey details, the materials, coded topics, and the statistics were all made publicly available in the companion website (Silva, 2013) and in Appendices B to E.

The remainder of this chapter is organized as follows: Section 4.1 discusses related work; Section 4.2 describes in detail the study design; Section 4.3 presents and provides discussion about the study results; Section 4.4 points out threats to validity; and Section 4.5 presents the conclusion.

4.1 RELATED WORK

Etzkorn et al. (Etzkorn; Delugach, 2000) compared various cohesion metrics with ratings of two separate teams of developers over two software packages. Their goal was to determine which of these metrics best match human-oriented view of cohesion. The developers rated class cohesion on a scale from 0 to 1. The ratings were then statistically correlated with a set of well-known structural cohesion metrics. Their study differs from ours in many ways. First, one of our goals is to better understand through a qualitative approach how developers perceive module cohesion. Second, we do not aim at comparing developers' ratings with an exhausted list of similar cohesion metrics. Rather, we analyzed two metrics representative of two distinct ways of measuring cohesion (structural and conceptual cohesion). Third, instead of rating cohesion through a numerical scale, we asked participants to rate which class was more cohesive in a given pair of classes. Also we asked them to explain their reasoning by means of open questions.

Similarly, Counsell et al. 2006 presented a study involving twenty-four subjects drawn from IT experienced and novice groups of developers. Subjects were asked to rate, in scale from 1 to 10, cohesion of ten classes sampled from two systems on a controlled classroom environment. Three research hypotheses guided their study, which involved quantitative analysis. Two hypotheses addressed whether or not cohesion as perceived by developers associates with class size and comment lines. The third hypothesis evaluated whether there is a noticeable difference between the ratings of experienced developers and novice ones. Besides the research methodology, their study differs from ours as they compared cohesion ratings with two other class features: size and amount of comments. Interestingly, one of their findings stated that cohesion is a subjective concept involving a combination of class factors and raters experience rather than any single, individual class feature per se. In our study we address the investigation of what would be the developers perception of such subjective concept, without having in advance any specific module feature as candidate to be associated with cohesion.

There are other related studies which applied web-based surveys to explore how programmers rate different software quality attributes like readability (Buse; Weimer, 2010), complexity (Katzmarski; Koschke, 2012) and coupling (Bavota et al., 2013). These stud-

ies applied mainly quantitative analysis. However, qualitative inquiry provides an interesting way of building knowledge on how human subjects reason about software quality attributes. In our case, besides doing quantitative analysis, we mainly focused on a qualitative investigation of how developers reason about cohesion by coding participants' responses.

4.2 STUDY DESIGN

The research questions two and three also presented in the introduction guided this study:

RQ2. How do developers perceive module cohesion? How do they reason about it?

RQ3. To what extent do structural cohesion and conceptual cohesion measurements relate with how developers rate cohesion of modules?

In order to address these research questions we collected developers' opinion to understand their perception and analyze how they react when having to reason about module cohesion. At first glance, interviewing would fit well as a research instrument for collecting developers' opinion. However, this would constrain some important requirements for our study. Such kind of study needs to provide classes' source code to participants' analysis and give them reasonable time to express their opinion. Interviewing would intimidate them by the presence of researchers or by the pressure of time. Also, interviewing is not an effective choice for collecting a reasonable amount of data without large cost. Therefore, we decided to conduct a survey as a web-based questionnaire. This technique has become one of the primary instruments in software engineering research (Shull; Singer; Sjöberg, 2010), and it can be used for both qualitative and quantitative inquiry. Combining both types of research methods are the best way to find answers and to build a convincing body of evidence to support or reject research hypotheses (Seaman, 1999).

The following sections present in detail how the survey was structured and applied.

4.2.1 Survey Overall Structure

The survey was divided in three groups of questions according to three categories of information: (i) information about participant familiarity with the class cohesion concept; (ii) information about how participants rate cohesion by analyzing classes; and lastly, (iii) information about participants' profile. Figure 4.1 illustrates the flow of questions divided into these three groups. For simplicity, questions from the second and third groups were abstracted away in the figure as they are explained in detail in the following sections.

We found reasonable to define questions in this order as it is recommended to approach "the most important stuff first" (Seaman, 1999). Thus, we left participants' profile questions at the end, which is probably the less motivating part to answer. We started with approaching the participants' knowledge about cohesion. For this, we applied the "funnel shape" (Seaman, 1999). First we asked broader questions about cohesion definition. Then, we narrowed down to questions that forced participants to concretely apply their perceived concept of cohesion by comparing and rating classes.

It is important to note the decision point between first and third questions in the

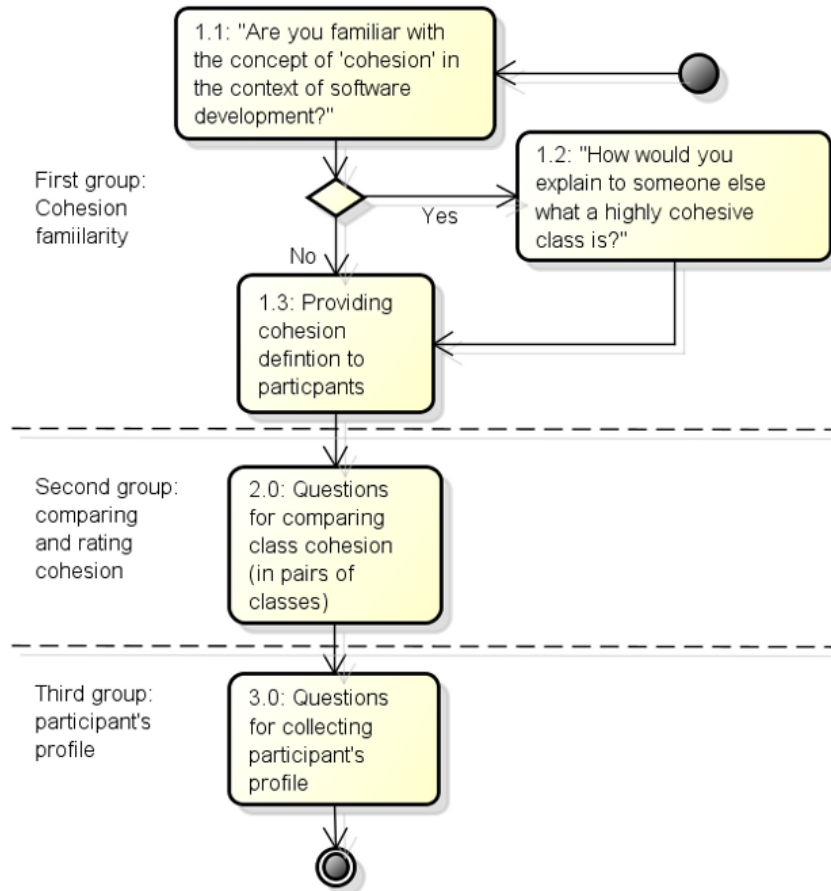


Figure 4.1 Questions flow overview.

first group (Figure 4.1). We only ask participants to explain what a highly cohesive class is if he or she declares him/herself familiar with such concept. Otherwise, we skip question 1.2 and jump to the cohesion definition (item 1.3). Item 1.3 is not a question. Instead, it is a set of statements defining what cohesion means. In this part of the survey, we were concerned about having a single page summarizing different flavors of cohesion definition extracted from different well-known authors in this field. We show this page on Appendix B. This would not bias the participant by presenting a partial view of cohesion. Moreover, for the sake of consistency, we showed this page to all the participants, even to whom declared him/herself familiar with the concept of cohesion.

4.2.2 Comparing and Rating Cohesion between two Classes

The second group of questions (see Figure 4.1) plays a central role in this survey. With them we aim at inducing participants to compare cohesion between two given classes and extract information about how they rate cohesion and how they explain their decisions. Given a pair of classes to be analyzed by participants we asked them to choose which class in the pair was more cohesive. Then we asked them to explain the rationale behind their

choice. All participants were asked to compare three pair of classes during the survey. In the following subsection, we explain why this number of pairs of classes and how we selected the classes. The companion website (Silva, 2013) presents a demonstration of the entire survey as it was applied. Also, Appendix B reproduces the survey questions as presented to participants.

4.2.2.1 Class Selection Criteria for Cohesion Comparison To address the second research question we carefully selected pairs of classes to be compared by the participants. We aimed to strategically define measurement scenarios that would allow participants to compare classes with high and low values of conceptual and structural cohesion. Moreover, we were concerned with avoiding confounding factors such as, classes with different sizes, naming conventions and comment density.

We ended up with three measurement scenarios, which guided us to select three pairs of classes.

Measurement scenario 1: *Both classes with similar conceptual cohesion measurement and distinct structural cohesion measurement.* The goal here is to analyze if participants' ratings match the difference of structural cohesion. This requires a pair of classes with the following characteristics: (i) classes equally cohesive in terms of conceptual cohesion, and (ii) one class much more cohesive than the other in terms of structural cohesion.

Measurement scenario 2: *Classes with opposite structural and conceptual cohesion measurements.* For example, if class *A* is more conceptually cohesive than class *B*, then *B* must be more structurally cohesive than *A*. This scenario exposes situations where the two different ways of measuring cohesion are contradictory. Thus we can analyze whether and to what extent participants' interpretation follow the structural or the conceptual measurement.

Measurement scenario 3: *Both classes with similar structural cohesion measurement and distinct conceptual cohesion measurement.* This scenario is similar to the first one but now what varies between the classes is the degree of conceptual cohesion. The goal here is, therefore, to analyze if participants' ratings match the difference of conceptual cohesion when structural cohesion is the same for both of the classes.

It is worth highlighting that participants did not have access to the classes' measurements in any part of the survey. Besides, we tried to minimize confounding factors that could affect participant's interpretation of class cohesion, as described as follows.

Size. It is a well-known confounding factor when analyzing module properties (Koru; Tian, 2005). So, we selected classes with similar LOC (Lines of Code). In addition, we filtered out too large classes that could be tiring and less motivating for participants to read and understand.

Context. We selected classes within a relatively simple context, which could be easily and shortly explained in a single survey page. Simple contexts do not require previous reading about the system the classes are related to.

Naming conventions. Participants might have more difficulty to understand classes whose source code is poorly named, without following any convention for naming iden-

tifiers. Thus, we selected classes with similar naming conventions. In very few cases, it was necessary to refactor some identifiers' names in order to balance the readability of the selected classes.

Source code comments. Finally, the presence of comments in the source code is another factor that may affect class comprehension. So, we selected pairs of classes with a balanced amount of comments.

Finding pairs of classes that match that set of requirements and measurement scenarios is not a trivial activity. We had to be very cautious as class comparison is a key part of the survey. Additionally, due to the subjectivity of some criteria we had to carry out this activity manually. That was challenging and took a reasonable effort when designing the study.

4.2.2.2 Cohesion Metrics In order to measure structural and conceptual cohesion we chose two metrics described in Chapter 2 – LCOM (Lack of Cohesion in Methods) by (Henderson-sellers, 1996) (also known as LCOM5) and LCbC (Lack of Concern-based Cohesion) (Silva et al., 2012), representing structural and conceptual cohesion measurements, respectively.

We selected LCOM5 for three main reasons: (i) it is available in an open source plugin for Eclipse IDE called Metrics¹; (ii) it is one of the variations of LCOM (the most known structural cohesion metrics family); and (iii) according to previous study (Chapter 3) it was present in a dimension of cohesion measurement together with other structural cohesion metrics such as LCOM3 and TCC, in most of the systems. Therefore, we consider LCOM5 as a good representative of structural cohesion metrics to be used in this study.

Regarding LCbC, we have been working with and studying this metric as well as other conceptual cohesion metrics in our research group for some years. LCbC is part of a concern-driven metrics suite which has been used in recent works (Garcia et al., 2005) (Figueiredo et al., 2008) (Figueiredo et al., 2012). Also, we are not aware of any other simpler and flexible metric for conceptual cohesion measurement. As explained in Chapter 2, the other conceptual cohesion metrics rely on text mining-based techniques for capturing information about concerns implementation. LCbC just counts the number of concerns a module addresses by using any desired strategy for concern mapping. As we worked with six classes (three pairs) in this study, it did not take long to manually identify and map concerns to classes' source code.

For practical reasons, we could not include more than one representative metric for structural and conceptual cohesion. As explained in Section 4.2.2.1, most of the study design effort was to strictly balance five confounding factors combined with two cohesion metrics and three measurement scenarios in order to select classes for the study. The addition of more cohesion metrics would make it very difficult to manually fulfill all the class selection criteria in reasonable time.

¹The Metrics Eclipse plugin is available at <http://metrics.sourceforge.net/>

4.2.2.3 Selected Pairs of Classes After analyzing classes from several open source projects in well-known repositories such as SourceForge and Github, we ended up with three pairs of classes from two software systems written in Java. The first system is for managing hotels². It has features for controlling bookings, billings and guest data. From this system, we selected four classes for the first and second measurement scenarios, as detailed in Figure 4.2. We obtained the pair of classes for the third measurement scenario from the FamilyTree project³. This is an open source academic system used by other empirical studies (Mäntylä; Lassenius, 2006). The source code of all selected classes is available in Appendix C as well as at the companion website (Silva, 2013).

Figure 4.2 also shows the corresponding structural cohesion (LCOM5) and conceptual cohesion (LCbC) measurements for each class. In addition, to facilitate interpretation, Figure 4.2 indicates the corresponding measurement scenario for each pair of classes. For example, in the first comparison, “>LCOM5” indicates that the class in the left-hand side has a higher LCOM5 value and “=LCbC” indicates that both classes have the same LCbC value, which matches with the first measurement scenario. In the third comparison, both classes have quite similar LCOM5 measurement⁴, whereas the left-hand side class has a higher LCbC value, thus the corresponding signals for this comparison are “=LCOM5” and “>LCbC”. Participants did not have access to this information.

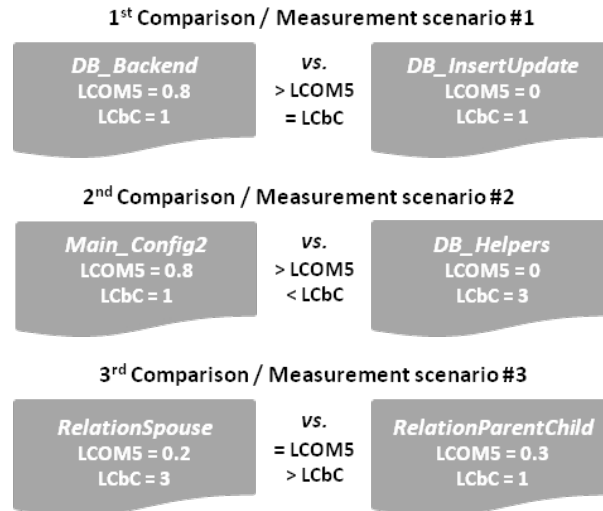


Figure 4.2 Classes for comparison.

²FGMP - Hotel Management, available at <http://sourceforge.net/projects/fgmp-hm/>

³FamilyTree project available at <http://www.soberit.tkk.fi/mmantyla/ISESE2006/>

⁴We considered a difference of 0.1 in LCOM5 as not sufficient to say both classes have different cohesion degrees according to this metric

4.2.3 Collecting Participant's Profile

Software engineers experience may influence on how they reason about software properties like cohesion. Thus participant's academic and professional background should be considered in this kind of study. However, there is no agreed way to collect programming experience data in empirical software engineering. Researchers have used distinct, sometimes not reported, ways of collecting and quantifying it. Some other researchers do not collect it at all (Feigenspan et al., 2012).

To build the third (and last) part of the survey questionnaire (see Figure 4.1 for the overview structure) we followed the results of a recent study on programming experience measurement (Feigenspan et al., 2012). It suggests a set of questions to measure participants' experience in software engineering empirical studies (Feigenspan et al., 2012). In Appendix B and in the companion website (Silva, 2013) we show all the questions we used for collecting participant's profile regarding professional and academic background.

4.2.4 Survey Sampling

The population for this study refers to software developers around the world with any experience on object-oriented programming and acting on any position in the software engineering field.

Any survey-based empirical study faces the problem of gathering participants, especially when dealing with such a broad population. That is why most of the survey-based software engineering studies use a non-probabilistic sampling (Seaman, 1999). This kind of sampling gathers participants who are easily accessible or the researchers have some justification for believing that they are representative of the population. Within non-probabilistic sampling, we referred to *Convenience* and *Snowball* sampling methods. The former consists of obtaining responses from people who are available and willing to take part. The latter involves asking people who have participated in the survey to nominate other people they believe would be willing to take part. We used snowball sampling only with some participants who showed interest in the study results, as we found they would be more willing to indicate other potential participants to be invited.

Some works have reported a low rate of participation, what has motivated recent investigation on this topic (Smith et al., 2013). We report the following factors and procedures we used to invite developers. They helped us to achieve a reasonable number of participants and improve our response rate. We apply here the same terminology used in (Seaman, 1999) (Punter et al., 2003) (Smith et al., 2013).

Personalized vs. Self-recruited survey. In self-recruited surveys, participants get to know somehow of the survey and decide to participate, while in personalized surveys each member of the sample is known and is personally invited to participate. The former can be easily spread by means of e-mail lists or social networks. However, researchers have no control of who participates in. We opted for a personalized survey to have control of who would be invited to participate. Thus, we were able to assure one response for each person and customize e-mail messages for invitations and reminders.

Reciprocity and Liking. People tend to comply with a request if they feel they owe the requester a favor (reciprocity) or if they have positive affect towards (liking). We

addressed reciprocity by inviting people who had invited us to participate in previous studies; and by inviting people who had some previous connection with us either by academic or by professional purposes. To address liking, in every invitation e-mail we tried to make a connection by using the person's name and first asking how they were feeling. We could confirm that people who had prior and stronger connection with the authors respond more than people who had subtle or no connection.

Authority and Credibility. Response rates may rise with the authority and credibility of the survey invitor. We approached this factor by emphasizing our affiliation as researchers/professors. The response rate might have risen as some respondents were former students of the authors or they had previously known our research laboratory.

Brevity. In general, the longer the survey takes to be completed the lower is the response rate. However, to keep empirical soundness, researchers have to carefully evaluate whether important questions can be removed. Thus, every time we prepared a new version of the questionnaire we made inspections to discuss and decide what could be removed or shorten in order to improve brevity. This is the main reason why we used just three pairs of classes to be compared by respondents.

Compensation Value and Likelihood. One of the strongest motivators is when the respondents feel that they get something back for their invested time. The return can come in many forms including monetary incentives, prizes, and gift cards. We provided a fifty dollars Amazon gift card for a drawing to participants who completed the survey. We highlighted this offer in every survey message such as invitations, reminders and welcome message.

Reminders. People easily forget to accomplish tasks they are not required to do. Therefore, we sent reminders to potential participants who had not answered the survey within a week. In addition, instead of programming automatic reminders, we decided to customize reminders messages according to whom we would send.

4.2.5 Data Collection

The data collection was preceded by a pilot study involving three participants with experience in software engineering industry and academia. We carried out the pilot study including all the expected steps for the survey execution (from invitation messages to data analysis). After analyzing their feedback and responses we made some adjustments on survey questions and presentation of the classes.

Afterwards, we executed the survey data collection. It lasted forty days, from February 23rd, 2013 to April 8th, 2013. We sent invitations in batches almost every day until March 03rd, 2013 by means of personalized e-mail messages. During data collection we monitored survey responses in order to have a crisp notion about how the web-based system were performing and how the respondents were acting in terms of their answers.

4.3 RESULTS AND ANALYSIS

In this section, we firstly describe the participants' profile and general statistics about who responded the survey. Then we present and discuss the results in the light of the

two research questions.

4.3.1 Participants' Profile

From 228 invitations sent to software developers around the world we had 34 incomplete responses (15%) and 80 full responses, representing a 35% of response rate. This number of participants is at least equivalent or superior to some recent studies in software engineering which used closed access survey (Bavota et al., 2013) (Smith et al., 2013). All the survey messages and the questionnaire were provided in English and Portuguese. We used the Portuguese version for Brazilian and Portuguese potential participants, although they were able to switch from the Portuguese to the English version. As expected, most of the participants were from Brazil and used the Portuguese version. In the end we had: 47 responses in Portuguese (58%), comprising one from Portugal and the others from Brazil; and 33 responses in English (42%) from 9 different countries (Canada, Germany, Chile, Japan, USA, Iran, Poland, Spain, and Brazil). The reminders played an important role as 33 responses (42%) were completed after we sent customized reminder messages.

Figures 4.3-4.6 summarize the participants profile in terms of occupation, academic degree and self-estimation on programming experience. Most of them declared themselves as software developers, system analysts, researchers and students. Some of them were software architects/designers and lecturers/professors. Few of them were testers, business analysts and project managers. Just one answered as unemployed or retired and other two answered as system administrator and software engineer. It is important to note that job position was a multiple choice question, so it was possible to have participants choosing more than one position (e.g., student and software developer). For this reason, we cannot know exactly how many participants were exclusively from software industry, but we can note that many of them have occupation in several different positions from software industry. In terms of academic degree, which was approached as a single choice question, two participants checked as undergraduate (2%), whereas nearly 31% checked as graduate (without post-graduate degrees). About 66% of the participants held some post-graduate degree.

We asked them to self-estimate their programming experience in two different ways as depicted in Figure 4.4. As explained in Section 4.2.3, we followed a set of questions to measure programming experience suggested by previous work on this topic (Feigenspan et al., 2012). First, we asked them to rate their experience on a scale from 1 to 10 (1 - very inexperienced and 10 - very experienced). Nearly 81% marked 7 or higher. Second, we asked them to compare their experience to colleagues' in a scale from 1 to 5. 65% declared themselves more experienced than colleagues by checking 4 or 5.

Regarding their experience with Java programming language (Figure 4.5), 12% checked 1 and 2, which means little experience, whereas 57% declared themselves well experienced (4 and 5 rates). With respect to object-oriented programming (OOP) (second chart of Figure 4.5), the ratings follow similar trend, with the difference that all participants declared themselves average experienced to very well experienced.

The boxplot charts in Figure 4.6 show, respectively, the overall number of years of programming experience and the number of years of programming experience in large

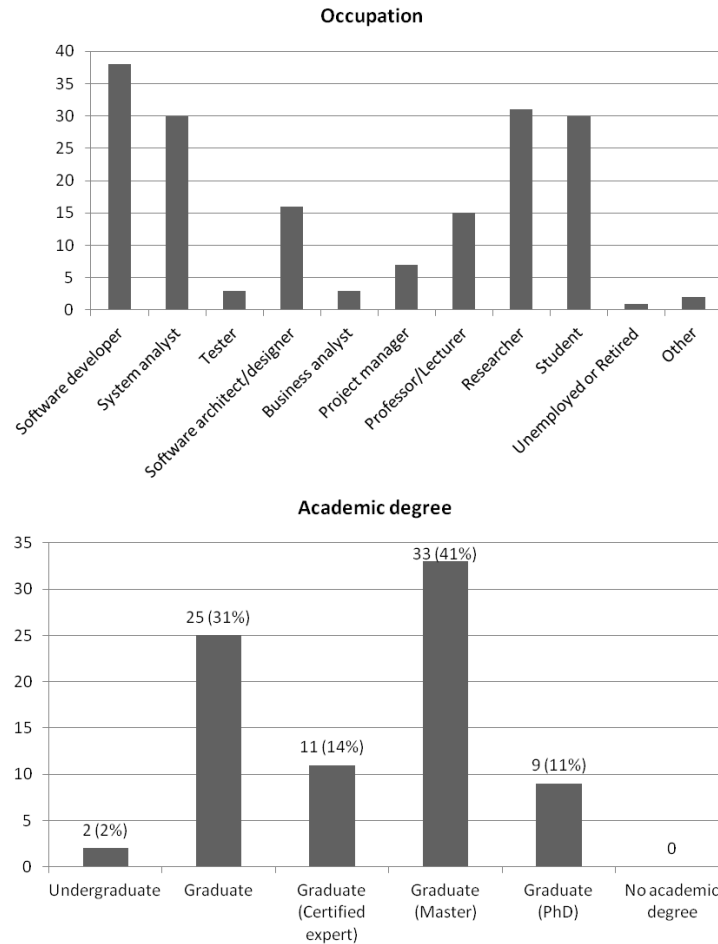


Figure 4.3 Participants' occupation and academic degree.

software projects. The plots are very similar, except that in the first one, experience varies from 1 to 29 years, whereas in the second one, experience varies between 0 and 15 years.

In summary, our set of participants includes developers with varied degrees of programming experience. We consider this result as positive for two reasons. First, it allowed us to make cross analysis between degrees of programming experience and the perception of cohesion. Second, the group of participants includes a representative set of developers with sufficient experience to answer the survey without difficulty.

4.3.2 How do developers perceive module cohesion? How do they reason about it?

To answer this research question (RQ2) we need to analyze answers of three parts of the survey, as presented in the following subsections.

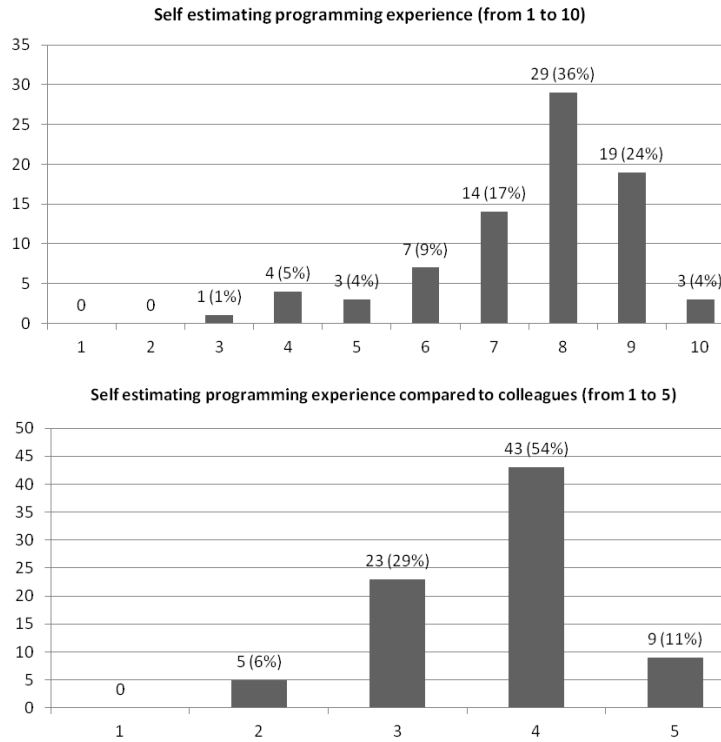


Figure 4.4 Participants’ self estimation on programming experience.

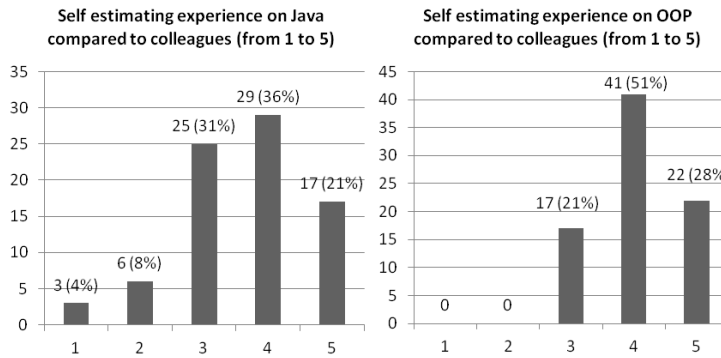


Figure 4.5 Participants’ self estimation on Java and OOP.

4.3.2.1 “Are you familiar with the concept of cohesion in the context of software development?” This is the first question of the survey (see Section 4.2.1 for the overall survey structure). From 80 respondents, 71 (89%) answered YES (familiar with cohesion) and 9 (11%) answered NO (not familiar with cohesion) for this question. Interestingly, most of the participants declared themselves as familiar with cohesion. Note that, at this point of the survey, no definition about cohesion was presented to the participants yet.

Having this result in mind, it is important to investigate whether cohesion familiarity

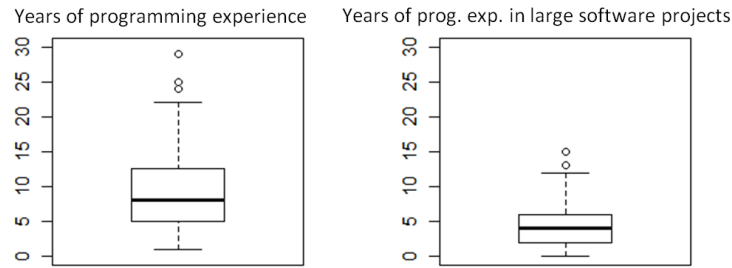


Figure 4.6 Participants’ programming experience in years.

is related with participant experience. To address this point, we cross-checked the survey first question responses with two other questions about participant’s profile as follows.

Cohesion Familiarity vs. Years of Programming Experience. As explained in 4.3.1, participants’ programming experience varies from 1 to 29 years. Additionally, we found that 35 participants declared having 10 years or more of programming experience. We consider these ones as the most experienced participants. They represent 44% of our sample. When analyzing their first question responses, we found that only two of them checked as not familiar with cohesion. The other seven not familiar with cohesion are spread over the participants with less than 10 years of programming experience.

Cohesion Familiarity vs. Academic Degree. We found that the distribution over academic degree considering only the 71 participants familiar with cohesion is very similar to this distribution considering all participants (see the second chart in Figure 4.3, which shows the distribution of academic degree for the entire sample). However, from the nine participants not familiar with cohesion, six are undergraduate or graduate programmers without post-graduate degree, whereas just three have Masters or PhD. Of these three, one has more than 10 years of programming experience but no experience in large software projects.

On the developers familiarity with cohesion. We can conclude that most of the participants of our sample are familiar with cohesion. Those who are not familiar with cohesion are most likely the least experienced ones.

4.3.2.2 “How would you explain to someone else what a highly cohesive class is?” To analyze answers for this question we coded responses from the 71 participants who declared themselves familiar with cohesion. Two members of our research group conducted the open coding process separately. One of them is co-author of this paper. Then, they had a meeting to cross-check, discuss and resolve conflicts in order to obtain the resulting coded responses.

As a result, we obtained 11 topics related to how participants explain what a highly cohesive class is. The distribution of these topics is shown in Figure 4.7. Each response may touch one or more topics as it is plausible to describe high cohesion by using a combination of correlated concepts. The top two topics are *class responsibilities* and *coupling*. Most of the participants (56) explain a highly cohesive class by using the

concept of cohesion in terms of class responsibility, for example: “*it is a class with a well defined scope of responsibilities*”; “*(...) It should not take responsibility for functions other than its own*”; “*(...) When it performs a well-defined role*”; “*Singular in purpose. It does one thing, and only one thing*”; “*All the included functionalities are conceptually highly related*”. This represents 78% of the 71 participants. We grouped in this topic similar terms like *features*, *concerns*, *functionality*, *role*, etc. Participants who touched this topic had a rationale and perception aligned with conceptual cohesion.

Thirteen participants mentioned the property of coupling in their explanation. This represents 18% of the 71 participants. Although high coupling does not necessarily mean low cohesion (and vice-versa), this is somewhat acceptable as we know that coupling and cohesion may be related to each other in many situations. Thus, some people prefer to explain cohesion by mentioning other concepts they might know better such as coupling.

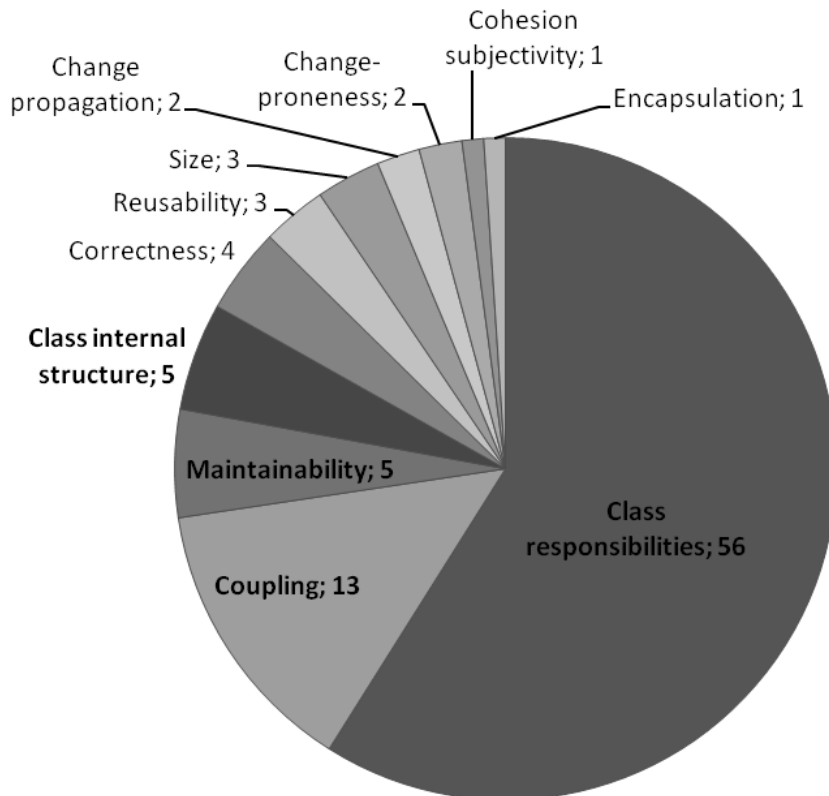


Figure 4.7 Coded topics from how participants explain highly cohesive classes (survey first question).

The *maintainability* topic, which we grouped with readability and comprehension, was mentioned by five participants (7%). Interestingly, some participants explained cohesion by mentioning external quality attributes such as maintainability, reusability and change-proneness. The presence of such topics is somehow understandable as cohesion may affect these properties.

In five responses (7%), participants explained high cohesion based on *class internal*

structure. They mentioned the relationship between methods and attributes within a class. For example: “Attributes and methods of this class have a strong logical relationship. I would say data-flow should be equally distributed within this class. If you think of a Def-Use-Graph almost all nodes should be connected to achieve high cohesion”; “Methods of this class implement features by calling each other”; “Its members are concerned mostly with themselves and other members of the same class”. This kind of responses reveals rationale and perception more aligned with structural cohesion.

Some not expected topics had few occurrences, such as *Correctness*. Curiously, four participants (5%) mentioned that cohesion is related to whether a class correctly fulfills its requirements. For them, cohesion degree depends on whether a class is correct or not with respect to what it is supposed to do. We made available all the coded responses in the companion website (Silva, 2013), including the ones from the other questions.

4.3.2.3 Rationale for Rating and Comparing Cohesion Additionally, we analyzed how participants reason about cohesion by investigating their rationale when rating and comparing cohesion between classes. As explained in Section 4.2.2, for each one of the three pairs of classes, we asked the participants to select the most cohesive class. Then, we asked them to explain the reasoning behind their choice. This was done in the second group of questions of the survey. To analyze their answers, we carried out an open coding process for their responses.

Regardless of the participants choice on which class was more cohesive, the most coded topic for all the three scenarios was *class responsibilities*. Again, most of the participants used the notion of class responsibilities to justify their choice when comparing cohesion between classes. From the 80 responses, for the first comparison scenario, *class responsibilities* was coded in 60 responses. *Class suitability to split* was the second most coded topic. It appeared in 15 responses. It was followed by *coupling* and *class internal structure* with 8 occurrences each. Turning to the second comparison scenario, conceptual cohesion perception (by means of *class responsibilities* topic) was found in 67 responses, *class suitability to split* in 17, *class internal structure* in 7, and *coupling* in 5 responses. Finally, for the third comparison, *class responsibilities* had 71 occurrences, *coupling* had 4, *class internal structure* had 2, and *class suitability to split* had just 1 occurrence.

Interestingly, we found the topic *Suitability to split* in the second group of questions to explain cohesion ratings but not in the first question – the one about highly cohesive class explanation (see Figure 4.1 for the questions overall structure). This is probably due to the set of statements defining the concept of cohesion we provided the participants after the first question. One of the statements say that a highly cohesive class should be difficult to split, according to Bieman and Kang (Bieman; Kang, 1995).

On the developers' perception about module cohesion. By analyzing how the participants explained class cohesion and how they reasoned when comparing cohesion between classes, we confirmed hypothesis that developers perceive cohesion in a conceptual manner, i.e., based on class responsibilities. Even though structural cohesion measurement has been mostly considered in academia and industry-scale tools, developers generally have a different perception about cohesion. We identified other ways of thinking about cohesion, which were coded during our analysis, such as coupling and structural cohesion. However none of them were as intensively used as conceptual cohesion perception was.

4.3.3 To what extent do structural cohesion and conceptual cohesion relate with how developers rate cohesion?

In order to answer this question (RQ3), we quantitatively analyzed the cohesion ratings for the three comparison scenarios. Also we qualitatively analyzed participants' explanation about their respective ratings. In addition, we cross-analyzed these data with cohesion familiarity and participant's experience to verify whether such factors associate with the results.

4.3.3.1 Cohesion Ratings and Respective Rationale Figure 4.8 shows the ratings distribution for the three scenarios of comparison. We applied the Fleiss Kappa statistical test (Fleiss, 1971) to quantitatively assess the degree of inter-rater agreement for each scenario, by using the R tool. This test is applicable as we had a fixed number of raters assigning nominal-scale ratings to a number of items. The Kappa coefficient κ lies in $[0,1]$ and indicates the level of agreement among raters: 1 indicates total agreement among all raters and 0 no agreement. As a result, for the first and third scenarios we found a slight agreement, with coefficients 0.193 and 0.155, respectively. For the second one we found fair agreement, with coefficient 0.234. This means that in first and third scenarios of comparison, the respondents slightly agreed to each other by means of rating in the same way, whereas in the second question their ratings fairly agreed. These categories (slight and fair agreement) were suggested by (Landis; Koch, 1977). Although it is used by some related works, for instance (Katzmarski; Koschke, 2012), they are not universally accepted. Literature recommends further analysis of the results to give support to any conclusions (Gwet, 2012), as we do in the following.

To better interpret the results of Figure 4.8 it is important to refer to Figure 4.2 to recall the measurement set corresponding to each scenario.

First scenario. In the first scenario, the second class (DB_InsertUpdate) is more cohesive than the first one (DB_Backend) in terms of structural cohesion, measured by means of the LCOM5 metric. On the other hand, they are equally cohesive in terms of conceptual cohesion, measured by means of the LCbC metric. Turning to participants' ratings, only 5% (4 participants) rated the second class as more cohesive. On the other hand, 37% (30 participants) rated both classes with quite similar cohesion, which matches with the conceptual cohesion measurement. Finally, 45 participants (57%) rated the first class as more cohesive, which does not match with structural cohesion neither with

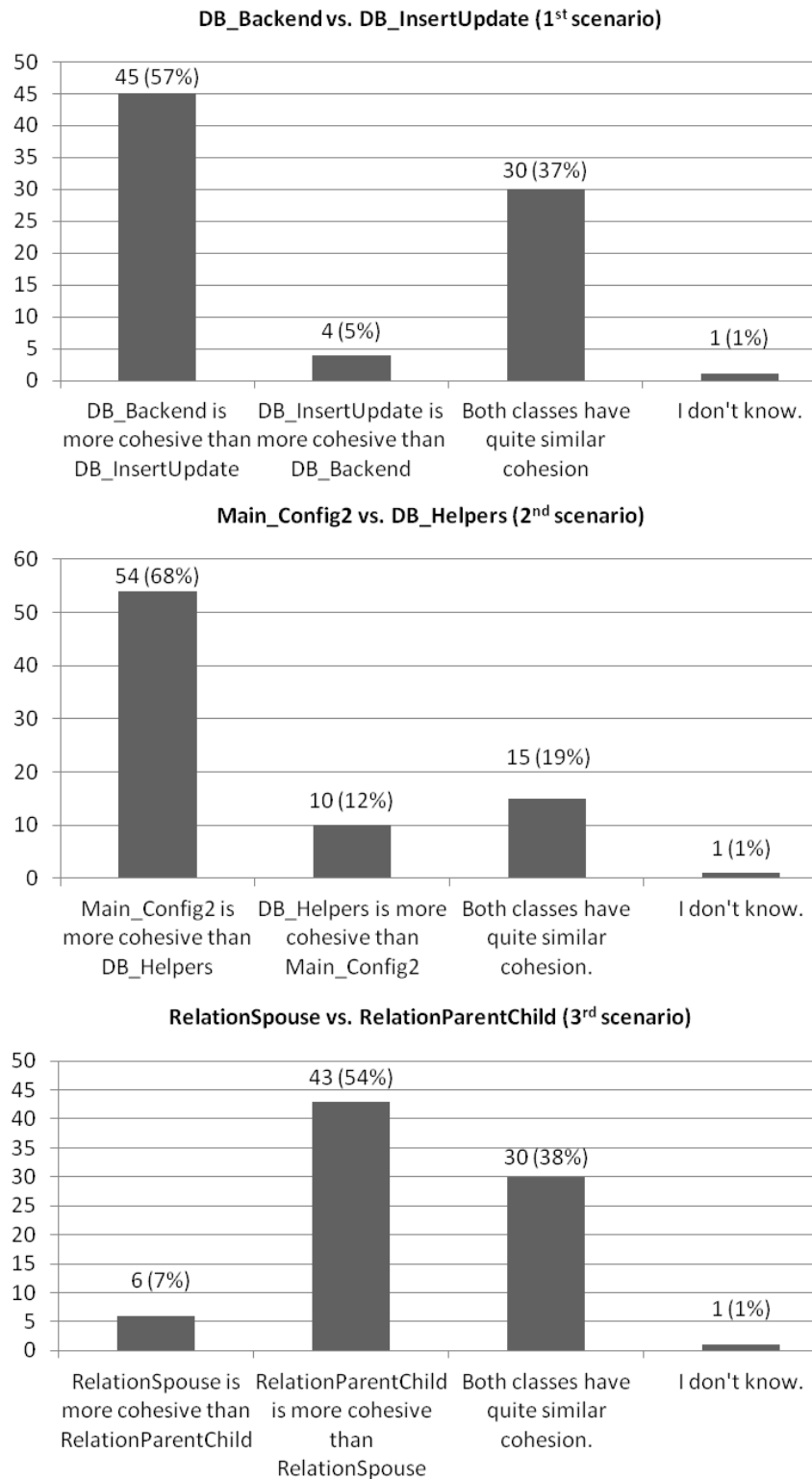


Figure 4.8 Cohesion ratings for the three scenarios.

conceptual cohesion. Therefore, analyzing quantitatively the amount of ratings for each category, we observe that most of the ratings do not match with structural cohesion measurement.

These results showed an issue to be further investigated – *Why did 45 participants rate the first class (DB_Backend) as more cohesive?* So, we turned to the responses where they explained their rationale. After analyzing the coded explanations, we found that of these 45 participants, 29 justified their ratings by using conceptual cohesion perception. Basically, most of them found the second class (DB_InsertUpdate) less cohesive because they considered that it contains two different concerns: inserting and updating information in database. Although this interpretation does not match with our initial measurement, as we considered inserting and updating as a single concern, called *data base writing*, it can be seen as correct. Other 12 responses mentioned that the class should be split into two, for instance: “(...) you could say that the Backend is more cohesive than InsertUpdate, as the second can be split into Insert and Update.”. This viewpoint is also associated with conceptual cohesion, as they claim the class should be split because it has multiple responsibilities. *Class coupling* were mentioned in 8 responses, whereas *class internal structure* were mentioned only in 7 out of the 45 responses. In summary, despite 57% (45) of the participants did not rate according to any of the target cohesion measurements, we found that most of them applied the conceptual cohesion perception to explain their reasoning.

We also investigated the reasoning of the four participants (5%) who chose the second class (DB_InsertUpdate) as more cohesive. At first sight, as it matches with the structural cohesion measurement, we would expect explanations related with this kind of cohesion perception. However, none of them used the structural cohesion view. Three found more responsibilities in the first class than in the second one. For example, one participant explained: “the first class handles database connections, which includes opening, closing and returning connections. The second one only updates the database”. *Readability and comprehension*, *complexity*, *suitability to split*, and *encapsulation* are topics with one occurrence each.

Finally, we analyzed the explanations from those who rated both classes with similar cohesion. From the 30 participants of this group (37%), 28 mentioned the *class responsibilities* topic and two mentioned the *suitability to split* topic. Other three topics had one occurrence each: *poor method decomposition*, *encapsulation*, and *class internal structure*. These observations indicate the association between the ratings of this group and conceptual cohesion measurement.

In summary, in the first scenario, most of the ratings matched with the conceptual cohesion measurement. Moreover, when analyzing the coded explanations, we found that most of them refer to the conceptual cohesion perception, even when the ratings do not match with it.

Second scenario. In this scenario, the first class (Main_Config2) is more cohesive than the second one (DB_Helpers) in terms of conceptual cohesion measurement, whereas the opposite stands for structural cohesion measurement. 54 participants (68%) rated the first class as more cohesive, which matches with the conceptual cohesion measurement. 15 participants (19%) chose both classes with similar cohesion. 10 (12%) rated the

second class as more cohesive, which matches with the structural cohesion measurement. Therefore, just analyzing quantitatively this result, we can conclude that most ratings associate with the conceptual cohesion measurement.

Analyzing the explanations of the participants who chose the second class as more cohesive (those that apparently matched with structural cohesion), we did not find any rationale referring to structural cohesion perception. Interestingly, all of these 10 participants gave explanations related with conceptual cohesion. 8 of them explicitly mentioned the identification of multiple concerns in `Main_Config2` class, for instance: “(...) *it contains DB information and billing information and currency information*”. Basically, they found that `Main_Config2` works with distinct and unfocused properties (billing, database, hotel information, etc.). The other 2 seem to have mistakenly checked the wrong answer – they explained the first class as more cohesive but picked the alternative “*second class is more cohesive*”.

The reasons for rating both classes as similarly cohesive were: *class responsibilities* (12 occurrences); *suitability to split* (6 occurrences); *coupling* and *class internal structure* with 1 occurrence each. In fact, 12 out of 15 participants justified their answers by mentioning class responsibilities. This shows that most of the participants who considered both classes similarly cohesive also had a conceptual cohesion perception.

Finally, we analyzed the explanations behind the 54 ratings that matched with conceptual cohesion measurement. 45 participants said that the first class (`Main_Config2`) was more cohesive because it was focused on a single behavior of the software (*class responsibilities* topic). 10 participants argued that this class is less suitable to split than the second one. This justification is somehow related to *class responsibilities*. Six participants justified their ratings by mentioning structural relatedness of class internal members. This shows that these six participants used the structural cohesion perception. Four participants applied the coupling concept as a proxy for cohesion, and one argued in terms of class comprehension. Overall, these results show that most of the participants used the conceptual cohesion perception. This reinforces the association between participants’ ratings and the conceptual cohesion measurement.

Third scenario. In this scenario, both classes are similarly cohesive in terms of structural cohesion measurement. However, in terms of conceptual cohesion measurement, the second one (`RelationParentChild`) is more cohesive than the first one (`RelationSpouse`). 43 participants, who represent 54% of the sample, rated the second class as more cohesive. These ratings match with the conceptual cohesion measurement. 30 participants (38%) rated both classes as similarly cohesive. These ratings match with the structural cohesion measurement. Six participants (7%) rated the first class as more cohesive. Therefore, it is clear to observe that most of the ratings associate with conceptual cohesion. However, further analysis is necessary to clarify other issues: *Why did 7% of the participants rate the first class as more cohesive? And why did 38% rate both classes similarly cohesive?*

Four of the six participants who rated the first class (`RelationSpouse`) as more cohesive seem to have done that by mistake. They explained their decisions as if the second class was more cohesive. The other 2 justified their decisions by mentioning the difficulty to understand the intention of the classes, and by using odd arguments like “*the first class is more cohesive because it makes several imports*”.

Apparently the 30 ratings for both classes with similar cohesion match with structural cohesion measurement. However, just two explanations behind them were actually related to structural cohesion perception. In addition, we coded 24 responses related to conceptual cohesion perception. However, 23 of them considered that both classes had the same number of concerns, thus assuming both classes similarly cohesive. For instance, one participant wrote: *“Both handles specific functionality”*. Only 2 mentioned structural cohesion in their reasoning, for example: *“I don’t see substantial differences in the usage pattern of members”*. Other 6 responses mentioned issues related to *coupling* (3 occurrences), *class similarity* (2 occurrences) and *reusability* (1 occurrence). These results show, therefore, that most of the participants’ rationale was based on conceptual cohesion.

Finally, we analyzed the responses of the 43 participants who chose the second class as more cohesive, matching with the conceptual cohesion measurement. We found terms related to the *class responsibilities* topic in all of them. This indicates that these participants’ rationale is strongly aligned with conceptual cohesion.

On the association between participants’ reasoning and cohesion measurement. Most of the ratings in the three scenarios do not associate with structural cohesion measurement. Actually, participants rated cohesion mostly based on class responsibilities, which shows they used the conceptual cohesion point of view. Even the participants who rated differently from the conceptual cohesion measurement applied the conceptual cohesion perception for justifying their decisions. These results confirm hypothesis that, even though structural cohesion measurement has been considered in academia and industry-scale tools, the perception of conceptual cohesion is predominant, which was also observed when addressing the previous research question (see Section 4.3.2).

4.3.3.2 Cohesion Ratings vs. Cohesion Familiarity In this section, we assess whether familiarity with the concept of cohesion associates with the cohesion ratings the participants gave. To do that, we cross-analyzed the ratings with answers of the survey first question (see Section 4.3.2.1), which asks whether or not the participant is familiar with cohesion.

First, we applied the Fisher exact test, which is a statistical test used to examine the significance of the association between two kinds of classification (two categorical variables) – in our case the nominal-scale cohesion ratings and bivariate cohesion familiarity (Yes/No). With this test we aimed at verifying whether the following *null hypothesis* is rejected: *There is no association between participant’s previous familiarity with cohesion and participant’s cohesion ratings*. After applying the Fisher exact test we got the following p-values respectively for the three scenarios: 0.170, 0.175, and 0.054. Therefore, we could not reject the null hypothesis for any of them. Hence, it is not possible, according to this quantitative assessment, to assume that there is an association between participants’ previous familiarity with cohesion and participants’ ratings on class cohesion. Then, we turned to the coded responses to try to find any possible association between these two variables.

Just nine participants considered themselves not familiar with cohesion in the beginning of the survey. Thus, we analyzed the corresponding ratings and explanations of these nine participants for the three scenarios. Regarding the first scenario, none of them rated the second class as more cohesive, that is, none of their ratings matched with structural cohesion measurement. Of these nine, one participant checked the “*I don’t know*” alternative. Four of the ratings matched with the conceptual cohesion measurement, that is, the alternative “*both classes have quite similar cohesion*”. After analyzing the corresponding coded explanations, we found that, regardless of the ratings, most of the participants (5 out of 9) justified their answers using the conceptual cohesion perception. Just two applied the structural cohesion perception.

Turning to the second scenario, one participant chose the second class (DB_Helpers) as more cohesive, which apparently corresponds to lower structural cohesion measurement. However, the participant justified his or her choice by applying the conceptual cohesion perception. He or she identified two different concerns in Main_Config2 class but did not identify additional concerns in DB_Helpers. This scenario also had one participant who checked “I don’t know”. So, there are seven remaining responses. Analyzing the coded explanations of them, five used the conceptual cohesion perception, three mentioned class suitability to split, and two used the structural cohesion reasoning. It is worth recalling that we often found two or more coded topics in a single response.

Analyzing the third scenario, most of the participants (6 out of 9) considered both classes with similar cohesion, apparently matching with structural cohesion measurement. However, from these six, just two explained their choice based on the structural cohesion perception. The others (4) justified their choice based on similarity of classes’ responsibilities, as they did not find additional concerns on the RelationSpouse class. The participant who checked the RelationSpouse class as more cohesive actually did it mistakenly according to his/her explanation. He or she explicitly explained RelationSpouse as less cohesive by using the conceptual cohesion perception and pointing additional concerns on it. Besides, we confirmed that the two who checked RelationParentChild as more cohesive, matching with conceptual cohesion measurement, indeed used the conceptual cohesion perception.

On the association between developer’s ratings and previous familiarity with cohesion. Conceptual cohesion perception was predominant even among participants who declared themselves not familiar with cohesion in the beginning of the survey. The overall conclusion for this sample subset follows the conclusion for the entire sample presented in previous subsections. This indicates that no matter whether or not developers have previous familiarity with cohesion, if they are provided with a brief explanation of what cohesion means (in different views – structural and conceptual), they apply the conceptual cohesion perception. These results support the claim that conceptual cohesion seems to be more intuitive and closer to the human-oriented view of cohesion.

4.3.3.3 Cohesion Ratings vs. Participants’ Experience Another possible factor that could influence the cohesion ratings is the participant’s previous experience in

programming. As shown in Figure 4.6, we asked participants the number of years they have in programming (in general) as well as in programming on large software projects.

Following the same strategy for the previous analysis, we applied the Fisher exact test. However, in this case we had to transform the years of programming experience variable, which is continuous, into a categorical variable. Thus, we split the years of programming experience data into the following categories: 5 years or less (≤ 5), more than 5 years but less than or equals 12 ($> 5 \text{ AND } \leq 12$), and more than 12 years (> 12). The first and third ranges represent the lower quartile and upper quartile, respectively (see Figure 4.6), whereas the second category comprises the interquartile range. The *null hypothesis* for the Fisher test was: *There is no association between participant's cohesion ratings and participant experience in programming.* As a result, we could not reject this null hypothesis for any of the three scenarios, as we found the respective p-values 0.7, 0.8 and 0.5. Therefore, we turned to the coded explanation trying to find any possible association between these two variables.

Based on participants' programming experience, we split the participants from the sample into two groups: the least experienced and the most experienced developers based on the lower and upper quartiles of programming experience distribution (Figure 4.6). The least experienced developers were 21 participants with 5 years or less of programming experience. The most experienced ones were 20 developers with more than 12 years of programming experience. Then we counted the occurrences of the coded topics in the three scenarios as summarized in Table 4.1. The columns represent the three scenarios for both groups of developers (≤ 5 and > 12 years of programming experience), and the rows represent the coded topics from the participants' explanation. Regardless of their experience, most of the participants used the conceptual cohesion perception (see "Class responsibilities." row). Other topics were used but with very few occurrences compared with *Class responsibilities*. *Class internal structure*, for example, had one occurrence on each group of participants for each scenario of comparison.

Table 4.1 Programming experience and coded topics for cohesion ratings explanation

Topics	≤ 5 years			> 12 years		
	1st	2nd	3rd	1st	2nd	3rd
Class responsibilities	17	17	17	14	17	17
Suitability to split	4	6	1	3	1	1
Reusability/modularization	1	1	2	4	0	0
Class internal structure	1	1	1	1	1	1
Readability/Comprehension	1	1	1	0	0	0
Complexity	1	0	0	0	0	0
Coupling	0	0	0	2	3	1

In summary, we could not reject the null hypothesis which states that there is no association between participant's cohesion ratings and participant experience in programming, by using Fisher exact test. However, we could observe that the *class responsibilities* topic was mostly mentioned both by the least experienced and the most experienced developers. We made the same analysis considering participants' experience in large software

projects and the results are very similar.

On the association between developer’s ratings and experience. We can conclude that conceptual cohesion represents the most common way of reasoning about cohesion regardless of the developers’ experience on programming. This is another important finding which supports the claim that conceptual cohesion seems to be more intuitive and closer to the human-oriented view of cohesion.

4.4 THREATS TO VALIDITY

Construct validity. Threats to construct validity concern the question how we know we are really measuring the attribute we want to measure. There are many ways to measure participants’ ratings regarding class cohesion that may influence the results. We opted to treat developers’ cohesion ratings as a categorical (nominal scale) variable based on a class cohesion paired comparison, as in (Katzmarski; Koschke, 2012). This scale is less powerful for applying statistical tests. Instead, we could have provided a set of classes and asking participants to give a cohesion rate within a range (e.g. from 0 to 1) for each class, as in (Etzkorn; Delugach, 2000), or we could have used a Likert scale (from 1 - very low cohesion to 5 - very high cohesion). However, with our choice it is cognitively easier for participants to make a decision comparing two concrete examples without needing to give a numerical rate for each analyzed class. Although this is less powerful by means of statistical tests, this helped us to achieve more reliable results by means of qualitative analysis.

Another possible threat is related to the way we provided classes to participants’ analysis. Participants could only analyze the classes under study without having access to other project documents. However, we selected classes relatively easy to understand. Also, we shortly described the overall purpose of each class and their corresponding context in the project. This is the choice for simplifying, as much as possible, participants’ analysis task, allowing them to focus on what is most important.

To measure cohesion, we used two metrics – LCOM5 and LCbC – which influenced on two parts of our study: (i) in the class selection process for matching the measurement scenarios (Section 4.2.2.1); and (ii) in the comparison between developers’ ratings with cohesion measurement (Section 4.3.3). We explained the reasons for choosing these two metrics in Section 4.2.2.2. However, we could have used other metrics, for instance, LCOM3 or LCOM4 (Briand; Daly; Wüst, 1998) for structural cohesion, and C3 (Marcus; Poshyvanyk; Ferenc, 2008) or MWE (Liu et al., 2009) for conceptual cohesion. They would probably lead us to select other classes for the study in order to match the measurement scenarios. Nonetheless, our main focus relied on capturing how developers reason about class cohesion and how their opinions associate with two different ways of measuring cohesion (structural and conceptual), regardless of the specific metrics applied. Additionally, we claim that LCbC has a higher potential to measure conceptual cohesion than C3 and MWE, as these two metrics compute cohesion based on topics extracted from source code comments and identifiers by executing automated text mining techniques. There is not enough evidence to rely on textually mined topics as proxies for

class responsibilities.

Internal validity. Read and interpret class code is a quite subjective task and there are distinct confounding factors that may influence it. Thus, we carefully defined a set of criteria to select classes to be analyzed by developers, as explained in Section 4.2.2.1. Participants' experience may also affect how the code is interpreted and how cohesion is rated. We discussed the participants' profile in Section 4.3.1, where we presented the set of participants with varied but sufficient experience to analyze code and rate cohesion. Also, the varied participants' experience was considered positive to enable cross analysis between research questions and experience factors such as years of programming and academic degree.

Conclusion validity. This comprises lack of statistical calculations or misuse of statistical assumptions that leads to incorrect conclusions made by the researcher. This study is predominantly qualitative and most of the analysis relies on discussion over qualitative data (coded responses) and by using basic statistic description illustrated and summarized in charts. Additionally, in Section 4.3.3.1 we used the Fleiss Kappa test, which is a non-parametric statistical test, to quantitatively assess the level of agreement among participants when rating cohesion. Then, we used the Fisher exact test (also non-parametric) in two situations: (i) in Section 4.3.3.2 to find a possible association between participant's previous familiarity with cohesion and participant's cohesion ratings; and in Section 4.3.3.3 to test the association between cohesion ratings and categorized participant's programming experience. In none of these two situations we could reject the corresponding null hypotheses, which led us to rely on qualitative analysis.

External validity. The main question related to external validity is whether the results discussed here can be considered to other groups of developers and classes of other systems. As discussed in Section 4.2.4, we applied convenience sampling instead of probabilistic sampling, as most of state-of-the-art software engineering papers do (Katzmarski; Koschke, 2012) (Bavota et al., 2013). However, our sample was not restricted to a group of developers with specific characteristic. As explained in Section 4.3.1, the analysis of the participants profile revealed an heterogeneous set of participants from nine countries with varied experience in programming and academic degree. Additionally, we limited this study to six Java classes from open source systems, mainly because we had to provide source code of only a limited number of class pairs to be analyzed in a relatively short amount of time. However, we excluded language features special to Java and the overall content of each class could be written in other object-oriented programming language.

4.5 SUMMARY

Although cohesion measures have been addressed in several works during the last decades, little is known about developers reasoning on module cohesion. The understanding of how developers perceive cohesion is important to know how they reason about such important quality attribute during software development and maintenance, as in most of the cases software modules are built and maintained by humans. This work provides empirical evidence on how developers perceive module cohesion and whether or not such perception associates with conceptual cohesion and structural cohesion – two state-of-the-art ways

of measuring cohesion.

The study involved a web-based survey with 80 participants from 9 countries and different levels of experience and academic degree. The survey comprised questions related to: general perception of module cohesion; module cohesion pair-wise comparison and rating; cohesion reasoning; and participant profile. We applied quantitative and mostly qualitative analysis through developer's responses. In summary, the results indicated that most of the developers are familiar with cohesion and those who are not familiar with cohesion are most likely the least experienced ones. Some developers use other popular concepts to explain cohesion such as "coupling" and "size" or more general concepts such as "maintainability" and "reusability". However, most of the developers perceive cohesion in a conceptual manner, i.e., based on class responsibilities, thus associating more with conceptual cohesion measurement. Moreover, this is the most common way of reasoning about cohesion regardless of the developers' experience on programming.

These results support the claim that conceptual cohesion seems to be more intuitive and closer to the human-oriented view of cohesion observed on software developers. Although structural cohesion measurement has been the most common way of measuring cohesion both in academic works and in commercial industry-scale software tools, our findings point out that conceptual cohesion measurement captures better the real notion of cohesion as perceived by developers in contrast with the traditional structural cohesion measurement. This result reinforces the need to improve knowledge and technology on conceptual cohesion measurement.

Particularly, in the next chapter, we analyze how conceptual cohesion metrics perform in the association with change-proneness in comparison with conventional structural cohesion metrics. We share Martin's view (Martin, 2003) that a cohesive module should be entirely focused on a single concern, because concerns are change drivers along each module's evolution history. Knowing which modules are more likely to change over time can improve the process of developing and evolving new or existing software modules in a given project by focusing attention on them.

CONCEPTUAL COHESION AND CHANGE-PRONENESS

The study presented in Chapter 3 showed that conceptual cohesion unveils an additional dimension of cohesion measurement. This result was supported by a quantitative assessment and explained by the underlying interpretation of cohesion which is based on the abstraction of concerns. It uses a different counting mechanism in comparison with structural cohesion metrics. Those findings gave us scientific evidence to support further investigations in order to understand how conceptual cohesion fits in comparison to conventional structural cohesion metrics.

One possible way to improve our understanding on the applicability of conceptual cohesion metrics is to assess their impact on software quality attributes. Software product metrics such as coupling, size and cohesion are often empirically assessed in terms of their impact on quality attributes such as fault-proneness, comprehension or maintenance effort, and change-proneness (Basili; Briand; Melo, 1996). The intention is to understand whether and to what extent a metric has the potential to be an indicator of a software quality attribute.

Therefore, in our research context, there is a lack of evidence to understand the impact of conceptual cohesion measurement on software quality attributes. We chose the *change-proneness* quality attribute due to concept of cohesion. Change-proneness (Madhavji; Fernandez-ramil; Perry, 2006) refers to the degree to which a software module is likely to change along its evolution history, whereas cohesion is the degree to which a module is focused on a single concern of the software. Hence, we share Martin (2003) hypothesis that the concerns a module implements may drive changes along the development and maintenance life-cycle. Thus, a highly cohesive module should focus on a single concern decreasing its likelihood to undergo changes, whereas modules addressing several concerns (low cohesion) are more likely to undergo changes. According to this view, concerns drive changes in modules' evolution history. It is a reasonable hypothesis as the more concerns a module addresses the more reasons it may have to change along the history. Knowing which modules are more likely to change over time can improve the

process of developing and evolving new or existing software modules in a given project by focusing attention on them. However, there is not enough empirical evidence to support this hypothesis. In this context, our fourth research question remains unanswered:

RQ4. *Whether and to what extent does conceptual cohesion associate to change-proneness?*

In addition, we also took the opportunity to compare the association between conceptual cohesion and change-proneness with the corresponding association between structural cohesion and change-proneness. It also supports us to understand whether the abstraction of concerns of conceptual cohesion metrics make it significantly different from structural cohesion metrics (RQ1).

Therefore, this chapter presents an empirical study to investigate the contribution of conceptual cohesion to the association between cohesion and change-proneness. We carried out an empirical assessment where we statistically analyzed whether and to what extent this association holds in comparison to other structural cohesion metrics. How strong is the correlation between conceptual cohesion metrics and module change-proneness? How do conceptual cohesion metrics perform in this correlation in comparison with structural cohesion metrics? Do conceptual cohesion metrics contribute to predict changes in modules when other structural cohesion metrics are also considered? This work supports the investigation about the association between conceptual cohesion and an external quality attribute, like change-proneness, in the context of industry-scale long-lived systems.

The remainder of this chapter is organized as follows: Section 5.1 explains study settings; Section 5.2 presents and discuss the results; in Section 5.3 the study threats to validity are then described; Section 5.4 discusses related work; and Section 5.5 presents the conclusion.

5.1 STUDY SETTINGS

The procedures for subject systems selection, concern mapping and cohesion measurement are the same from the previous study presented in Chapter 3. Table 5.1 presents additional information in the last three rows regarding the history analyzed for change counting. All the analyzed history totaled 30,248 revisions. The shortest period was 51 months (jEdit) and the longest period was 94 months (Rhino). Those periods were formed from the day when the selected release was delivered up to the day we made the change counting procedure (described in Section 5.1.1). So, for example, in JEdit system, the selected release (4.3.2) was delivered in May 2010, then we analyzed from this month up to July 2014, when we executed this study phase for all systems.

In next section we describe how the changes were counted for each module of the subject systems. Then, Section 5.1.2 presents the statistical methods applied for the analysis.

Table 5.1 Systems and analyzed change-set

	JFreeChart	Rhino	jEdit	Tomcat	Findbugs	Freecol
Release	1.0.6 (June/2007)	1.6R5 (Nov/2006)	4.3.2 (May/2010)	6.0.26 (Mar/2010)	1.3.5 (Sep/2008)	0.84 (Aug/2009)
Description	Chart library for the Java platform	Mozilla's Javascript interpreter	Text editor for programmers	web server container for the Java platform	Static analysis tool to find bugs in programs	A civilization-like game
LOC	76,059	59,182	109,516	161,735	98,914	75,902
# of Java files	514	156	531	1060	1041	431
Repository	SVN (Source-Forge)	Git (Mozilla)	SVN (Source-Forge)	SVN (Apache)	SVN (Source-Forge)	SVN (Source-Forge)
# of revisions	3,271	2,765	5,851	8,941	4,612	4,808
# of months analyzed	86	94	51	53	71	60

5.1.1 Change Counting

This study involves the Change Count (CC) metric computation, which quantifies the amount of changes each module suffered along its evolution history. This metric counts the number of revisions on which each module was changed. For instance, if a module m was changed in 10 revisions throughout the analyzed period of time then $CC(m) = 10$. We considered each commit in the source code repository as a revision. It is important to highlight that each commit in the used source code repository involves all the Java files changed in the revision. Table 5.1 presents the number of revisions and number of months we took into account for each system. We considered the period of time from the selected releases until July 2014 (the month when the data collection was finished). We used CC as the dependent variable when analyzing the association of the cohesion metrics and change-proneness.

After the change counting step, we merged all the measurement results in a single file, for each system, to be used as input in the R statistical tool¹. All the measurements are available at (Silva, 2015a). In summary, we gathered values for eight metrics: five structural cohesion metrics, two conceptual cohesion metrics, and the change count metric. All these metrics were computed per module. Given the 3,733 Java files, this totaled about 29,000 data points.

¹Available at <http://http://www.r-project.org/>

5.1.2 Method of Analysis

The study research question is concerned with investigating whether and at what extent conceptual cohesion associates to change-proneness. Additionally, we address how this association fits in comparison with conventional structural cohesion metrics. Therefore, we first applied a correlation analysis to evaluate how strong is the correlation between conceptual cohesion metrics and module change count. Also, we compare the performance of such correlation with the performance of conventional structural cohesion metrics. For this purpose, we applied the Spearman correlation method (Cohen, 1988), which is a non-parametric method, as the data distributions were non-normal. We made the distributions available at (Silva, 2015a).

In addition to the correlation analysis we used a machine learning technique, called Regression Tree (Breiman et al., 1984), to support us on finding evidence about the association between conceptual cohesion and change-proneness. With Regression Trees we built tree-based models to co-identify highly-changed modules and those with the highest cohesion measurement values. This method attempts to establish predictive relations through recursive partitioning. In tree-based models, results are represented in tree structures. Each node in a tree represents a set of data points which is recursively partitioned into smaller subsets, as illustrated in Figure 5.1. The data used in such models consist of multiple attributes. One attribute should be identified as the response variable, which is change count in our case; and one or more other attributes identified as predictor variables, which are the conceptual and structural cohesion measurements in this study.

Figure 5.1 illustrates an example of a regression tree for JFreeChart system. Each node represents a group of modules, where we show: (i) the cutoff variable, which is the cohesion metric found by the regression algorithm that best cuts the sample to achieve the maximum deviance reduction in change count; (ii) the change count average of the modules within a group; and (iii) the group size in terms of number of modules as well as the corresponding percentage with respect to the total number of modules analyzed.

The regression algorithm starts with the complete data set and recursively partitions it into two smaller subsets. In the example, LCOM2 was selected by the algorithm to partition the modules with cutoff value of 562. Thus, modules with LCOM2 less than 562 are placed in the left-hand side partition, which groups 95% of JFreeChart modules having CC value of 6.9 in average. Whereas, modules with LCOM2 equals or greater than 562 are then placed in the right-hand side partition with $CC = 24$ in average and thus representing the top 5% most changed modules in average. The left-hand side group of 490 modules is then partitioned by using LCbC_XScan metric following the same algorithm. For each new partition the regression tree algorithm evaluates whether or not it has to partition again. Each partitioning performed by the algorithm finds the most homogeneous two groups in terms of change count because, for each group, the difference between the average change count and the actual change counts for individual modules was minimized.

With the complete tree built, it is possible to see which cohesion metrics ended up in the regression model as change count predictors. Following the example, LCOM2, LCbC_XScan, LCOM4 and TCC were the selected response metrics for predicting change

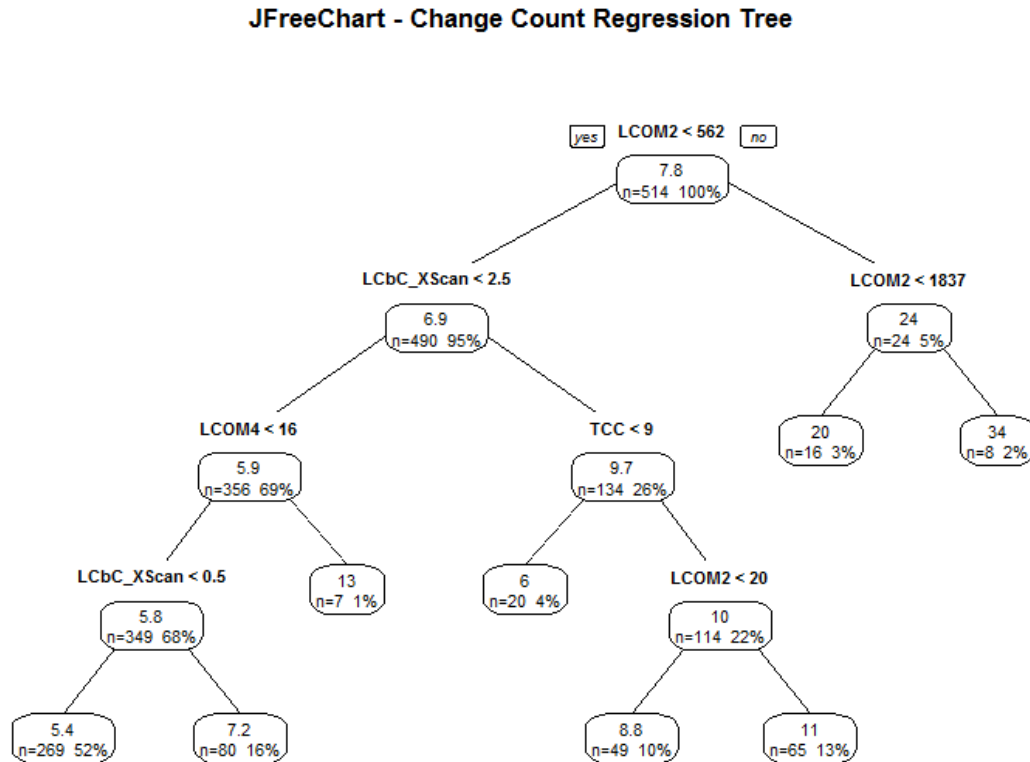


Figure 5.1 An example of a regression tree having CC as response variable

count. If one needs to predict the 30% of the JFreeChart modules that most changed, the following reasoning applies over the resulting tree: first, select the modules with LCOM2 equals or greater than 562 as it represents the 5% most changed modules; second, with the remaining 95%, follow the path of $LCOM2 < 562$ and $LCbC_XScan \geq 2.5$ to obtain the 26% most changed modules besides the other 5%.

The R scripts for correlation tests as well as for generating and plotting regression trees are available in Appendix F.

5.2 RESULTS AND DISCUSSION

This section focus on presenting results and analysis after applying correlation test and regression tree in order to address the fourth research question. We also discuss typical situations that might affect positively and negatively the association between conceptual cohesion and change-proneness.

Correlation Analysis. We show the results for the correlation tests in Figure 5.2. Each cell shows the Spearman correlation coefficient between each cohesion metric and change count for each system under study. In order to give a qualitative label for the correlation coefficients, we follow Cohen's range (Cohen, 1988), which suggests that a

coefficient from 0 to 0.30 is a weak correlation; from 0.30 to 0.50 is a moderate correlation; and higher than 0.50 is a strong correlation. Some correlation tests did not achieve a significance level, having a p-value higher than 0.05, thus we represented such cases with a “*” character in Figure 5.2.

	JFreeChart	Rhino	jEdit	Tomcat	Findbugs	Freecol
LCOM2	0.5	0.65	0.17	0.38	0.48	0.48
LCOM3	0.38	0.37	0.16	0.32	0.37	0.21
LCOM4	0.33	0.32	*	0.21	0.22	0.21
LCOM5	0.2	0.31	0.2	0.27	0.33	0.24
TCC	0.25	*	0.19	0.21	*	0.28
LCbC_Xscan	0.63	0.32	0.18	0.33	0.21	0.47
MWE	0.26	0.24	0.13	0.10	*	-0.23

* no significance level achieved

Figure 5.2 Correlation between each cohesion metric and change count

Figure 5.2 also highlights the highest coefficients in each system. This shows that, on average, LCbC_XScan and LCOM2 are the most correlated with change count. LCbC had strong correlation in JFreeChart, whereas LCOM2 had strong correlation in Rhino. In jEdit all metrics had weak correlation. In Tomcat, LCOM2, LCOM3 and LCbC had almost the same coefficient achieving a moderate correlation. In Findbugs, LCOM2 had a moderate correlation. In Freecol, LCOM2 and LCbC had almost the same correlation at moderate level. Some metrics such as MWE and TCC had a weak correlation with change count in all systems.

In summary, although LCbC was not always the most correlated with change count, this metric together with LCOM2 varied between the most and the second most correlated in five of the six systems. When comparing the two conceptual cohesion metrics LCbC outperformed MWE. Overall, these results suggest that it is possible to have a conceptual cohesion metric moderately or strongly associated to change-proneness. Among the metrics we analyzed, LCbC and LCOM2 are the most promising indicators of change-proneness from the cohesion perspective.

Regression Tree Analysis. Besides the regression tree illustrated in Figure 5.1 we show the generated trees for the other systems in Appendix G. They follow the same drawing pattern, having each node showing: the cutoff metric and corresponding cutoff value; average change count of the corresponding group; and absolute and relative group size.

Overall, LCOM2 was the only metric present in all regression trees. LCbC_XScan was present in Tomcat, Freecol and JFreeChart, while MWE was present in Tomcat, Findbugs and Freecol.

It is out of scope of this study the search for the best prediction model for change-proneness. Differently, here we aim at analyzing if conceptual cohesion metrics contributed to the final regression model when using only cohesion metrics as predictor

variables. Certainly the best prediction models would include other metrics for different properties such as coupling and size. However, the inclusion of such metrics in this study would shift the focus from comparing cohesion metrics.

Therefore, we can conclude that it is possible to have conceptual cohesion metrics contributing to predict change-prone modules together with other cohesion metrics. LCbC_XScan and MWE were present in four out of six systems analyzed. This result confirms correlation analysis and gives us statistical evidence that there is an association between conceptual cohesion and change-proneness and that such association, for some analyzed systems, performs equal or better than the association of structural cohesion metrics and change count.

Additionally, in the context of our study we tried to understand possible cases that might have increased and decreased the association strength between conceptual cohesion metrics and change-proneness. Therefore, we went deep into our data set to analyze some classes and their evolution history in order to better understand typical cases where conceptual cohesion metrics scored and failed in the association with change count.

Success depends upon the concern mapping. We used the same subject systems and cohesion data set from the previous study (Chapter 3). Therefore, the same constraint related to concern mapping applies to the current study. In general, conceptual cohesion succeeds on indicating module cohesion and on its association to change-proneness whenever the underlying concern mapping succeeded on identifying concerns in modules. Therefore, besides other factors that may influence on the potential of conceptual cohesion as a change-proneness indicator, we could observe that concern mapping accuracy is determinant. This supports us on investing effort to investigate the impact of varying concern mapping strategies for conceptual cohesion measurement other than analyzing other possible factors.

For instance, the *StandardManager* class in Tomcat is responsible for managing the pool of sessions in Tomcat web container, having 8 concerns identified by XScan. The eight concerns place it at the Top 3% least cohesive class according to LCbC measurement. This class changed 33 times, thus ranked at the Top 5% most changed classes in Tomcat. We manually analyzed the change history of this class and found that indeed many changes were related to the eight concerns identified by XScan and computed by LCbC. Turning to structural cohesion, we take LCOM2 metric to represent this group of cohesion metrics as it had the best correlation with change count, in general. The *StandardManager* class had LCOM2 = 158, which places it at Top 13% least cohesive class according to this metric. As a consequence, in such cases LCbC is closer to change count, which make it scores better than LCOM2 in such comparative association.

We also identified a similar situation in classes of all other analyzed systems. For example, the *ReportColonyPanel* in Freecol, according to LCbC, was at Top 10% least cohesive class, while according to MWE, it was at Top 14% least cohesive class. This class changed 62 times, then placed at the Top 10% most changed class. We manually identified several changes related to the concerns implemented by this class. According to LCOM2 it was at Top 31% least cohesive class. As a result, it is another case that explains typical situations where conceptual cohesion metrics worked out as better indicators of change-proneness.

However, as also reported in Chapter 3, we found some classes where the underlying concern mapping of conceptual cohesion metrics failed to identify concerns. We observed classes with few or no concerns mapped by the underlying mapping strategy while we manually identified concerns that should have been computed. In addition, in some situations where conceptual cohesion failed on computing the classes actual concerns, structural cohesion performed well on identifying lack of cohesion and scoring on the association with change-proneness.

For example, the *JEditBuffer* class represents the content of an open text file in jEdit user interface. This class had the 4th highest LCOM2 value as well as the 6th most changed class in the project. According to these metrics, *JEditBuffer* is at the Top 1% least cohesive rank and among the Top 1% most changed classes. However, according to conceptual cohesion metrics, it is highly cohesive. LCbC was 0, as XScan was not able to identify important concerns implemented by this class, neither the main concern of text editing. Also, MWE ranked this class at Top 60% least cohesive classes. This is a typical situation found in all systems we analyzed which contributed to minimize the association between conceptual cohesion and change-proneness, whereas it scored in favor of structural cohesion metrics. Although this observed phenomenon hindered the performance of conceptual cohesion in our study, we still found substantial results to support our hypothesis and to conclude that conceptual cohesion is worth to be further investigated as a promising way of cohesion measurement.

5.3 THREATS TO VALIDITY

This current study used the same subject systems and data set from the study in Chapter 3. Therefore, the threats to validity discussed in Section 3.3 can also be applied here. In addition, we discuss two specific threats related to internal validity.

When computing Change Count, we did not filter the revisions based on the type of changes. We simply considered every add, delete and modify operations as changes regardless whatever motivated them. So, we took into account changes of any type such as bug fixing, inclusion of new features, preventive maintenance or refactoring. As a consequence, we did not eliminate the noise represented by non-essential changes throughout the software history we analyzed. For example, moving operations among repository folders or updating comments for license changing. To overcome this threat we adopted two requirements on selecting the six analyzed systems: (i) we searched the systems website to find intermediary releases reporting a reasonable number of changes in their later releases; and (ii) a candidate system release should have a repository with a large change scenario ahead in their commit history. Table 6.1 shows that we analyzed commits along 51 months (jEdit) to 94 (Rhino). Therefore, even in the presence of noise the change sets were predominantly composed by essential changes.

Another issue is regarding the regression tree analysis. We are aware that cohesion is not the only internal quality attribute that affects module change-proneness. To build complete models for change-proneness prediction other attributes have to be considered. However, this is out of scope of this study as our purpose was to investigate whether there is an association between conceptual cohesion and change-proneness. Therefore,

we applied a prediction modeling technique in order to evaluate how conceptual cohesion metrics fit together with structural cohesion metrics as explanatory variables having change count as response variable. We were not concerned on finding the best predictors for changes in modules. Instead, we were concerned on analyzing whether conceptual cohesion metrics contribute to predict changes in modules when structural cohesion metrics are also considered in the same conditions.

5.4 RELATED WORK

Several researchers have investigated the association between one or more cohesion metrics and quality attributes, such as fault-proneness, changeability or change-proneness. In this section we briefly present how recent work have addressed this issue, what are their findings, and how our investigation fit in this scenario.

Dallal and Briand (2012) assessed the relationship between structural cohesion metrics and fault-proneness. They included their new cohesion metric called Lack of Similarity-based Class Cohesion (LSCC), which computes cohesion based on the interaction degree among methods within a class. Their results suggest that class quality, measured in terms of fault occurrences, can be more accurately explained by cohesion metrics that account for the degree of interaction between pairs of methods, such as LSCC. Their study did not include conceptual cohesion metrics.

Kabaile, Keller and Lustman (2001) studied the relationship between two structural cohesion metrics (LCC and LCOM) and changeability. Changeability is perceived as a quality indicator of the capacity of a software module to absorb changes. Their study involved three object-oriented systems in C++, and their results did not point any relationship between those two metrics and changeability that justified the application of this metric as changeability indicator.

Chowdhury and Zulkernine (2010) studied the relation of several structural metrics for OO systems, including LCOM2, with software vulnerabilities. They found that LCOM2 moderately correlated to vulnerabilities, and they suggest that this metric can be used, together with complexity metrics, as early indicators of vulnerabilities in software.

Koru and Tian (2005) identified and compared modules that change most and modules with the highest structural measurement values of several properties in two large-scale open-source projects. Among other metrics, they included six for structural cohesion. They did not find a strong match between top modules in change count rankings and top modules in rankings of structural metrics, including cohesion. Koru and Liu (Koru; Liu, 2007) also conducted a study over the same two projects to characterize change-prone classes, but only one cohesion metric (LCOM3) was considered in this study. They found that cohesion appeared in the regression models as secondary feature.

Romano and Pinzger (2011) investigated the correlation between change-proneness and cohesion for Java interfaces. They found that the Interface Usage Cohesion (IUC) metric, also proposed in that work, exhibited the strongest correlation with source code changes. They also proved that IUC is an adequate metric to compute prediction models, based on machine learning algorithms, for classifying change-prone Java interfaces. Their study did not address other kinds of modules than interfaces.

Finally, Lu et al. (2012) also assessed the ability of OO metrics to predict change-proneness. Their study included 18 structural cohesion metrics. They found that cohesion metrics generally have a lower predictive ability compared to size metrics.

The metrics named C3 (Marcus; Poshyvanyk; Ferenc, 2008), CLCOM5 (Ujhazi et al., 2010) and MWE (Liu et al., 2009), that we presented in Chapter 2, were empirically evaluated as fault-proneness indicators together with structural cohesion metrics. In summary, their results indicate that they can be used together with structural cohesion metrics to improve the prediction of faults in classes.

In summary, although some studies have investigated the relationship between structural cohesion metrics and external quality attributes, to the best of our knowledge there is no work that has studied the relationship between conceptual cohesion metrics and change-proneness. Actually, as far as we know, (Marcus; Poshyvanyk; Ferenc, 2008), (Ujhazi et al., 2010) and (Liu et al., 2009) are the only works that empirically studied the association between conceptual cohesion metrics and a quality attribute such as fault-proneness. However, the authors observed that these metrics strongly depend on comments and naming conventions applied in the source code. When they are missing, these metrics do not perform well. Moreover, code refactoring for applying naming conventions and comments is not a trivial task and may not be cost effective.

5.5 SUMMARY

This study is a continuing work on the quest for providing empirical evidence about the contribution of conceptual cohesion metrics. Previous study presented in Chapter 3 gives statistical basis for showing that conceptual cohesion is an significant additional way of cohesion measurement. In other words, conceptual cohesion represents an additional dimension of cohesion measurement which is not captured by conventional structural metrics, and that it is worth continuing research effort. The study presented in this chapter complements the previous one by showing evidence that such additional dimension captured by conceptual cohesion metrics adds to structural cohesion when they are considered together for predicting changes in modules. Our correlation analysis also confirmed that conceptual cohesion, represented by LCbC, performed well as an indicator of module change-proneness when compared to the performance of structural cohesion metrics. Additionally, we highlighted and discussed typical situations where conceptual cohesion metrics worked well or not in the association with change-proneness. Regardless other factors that may influence, this basically depends on how well the underlying concern mapping strategy identifies concerns addressed by the analyzed classes.

In summary, the results have shown that conceptual cohesion can leverage the association between cohesion and change-proneness. This reinforces conceptual cohesion as an additional way of cohesion measurement and that it is a promising approach that is worth to be further considered in software measurement research and practice. As we empirically observed in previous and current chapters that the success of conceptual cohesion strongly depends on concern mapping, in next chapter we investigate the impact of different concern mapping strategies on conceptual cohesion measurement.

THE IMPACT OF DIFFERENT CONCERN MAPPING STRATEGIES ON CONCEPTUAL COHESION

Several researchers have attempted to provide an objective and effective way to measure module cohesion (Briand; Daly; Wüst, 1998). Most of them rely on structural information extracted from the source code, as we explain in Chapter 2. In contrast, there is an alternative group of recently proposed cohesion metrics which attempts to measure module cohesion by extracting information about concerns addressed by the modules (Marcus; Poshyvanyk, 2005) (Liu et al., 2009) (Silva et al., 2012). Also, we describe them in Chapter 2. In Chapter 3 we provide empirical evidence that metrics in this group have an additional dimension of cohesion measurement that is not captured by conventional structural cohesion metrics. Moreover, conceptual cohesion is closer to how developers reason about module cohesion (Chapter 4).

However, measuring conceptual cohesion is not straightforward as it is difficult to capture what concerns a module realizes. We summarized concern mapping techniques in Section 2.2.3. In addition, in previous chapter we discussed some problems related to the concern mappings we used for measuring conceptual cohesion through LCbC and MWE. We analyzed some modules and could find that the underlying techniques for concern mapping might have decreased the correlation degree between conceptual cohesion and change-proneness. However, at that point we did not have enough evidence to support this observation. Moreover, we did not know how and at what extent different strategies for concern mapping affect conceptual cohesion measurement and its impact on change-proneness.

Therefore, the main goal of this study is to provide empirical evidence of whether, how and at what extent different concern mapping strategies impact on conceptual cohesion measurement. In addition, we also analyze whether differences on concern mapping strategies impact on the association between conceptual cohesion and change-proneness. To achieve this goal we stated the fifth, sixth and seventh questions of our research scope also presented in Chapter 1:

- RQ5: Do different concern mapping strategies impact on conceptual cohesion measurement?
- RQ6: If such impact is significant, can we explain how different is conceptual cohesion over different mapping strategies?
- RQ7: Does it influence on the correlation between conceptual cohesion and change-proneness?

We carried out an empirical study where we investigated how conceptual cohesion varied over three different concern mapping strategies having six systems under analysis. In summary, we could provide empirical evidence that: (i) different concern mapping strategies significantly impact on conceptual cohesion measurement; (ii) manual mapping increases the correlation between conceptual cohesion and change-proneness in contrast to other two automatic strategies we used; (iii) as manual mapping strategy is subjective it can drive different cohesion distributions depending on what auxiliary artifacts developers use to carry out mappings; (iv) although the manual strategy outperformed the automatic ones, it is also possible to have automatic mappings driving moderate to strong correlations between conceptual cohesion and change-proneness; and (v) one of the two automatic strategies we analyzed should be considered when manual mapping is not possible to be applied. Additionally, as another contribution of this work, we provide to the research community a manual concern mapping carried out by us over jEdit system and a description about our manual mapping process. The purpose is to make it available for other studies as it is not easy to find manual mappings intended to cover as much as possible the system concerns and modules.

The remainder of this chapter is organized as follows: Section 6.1 describes the study design; Section 6.2 presents results and findings; Section 6.3 discusses threats to validity; Section 6.4 describes related work; and Section 6.5 presents the conclusion and future work.

6.1 STUDY SETTINGS

The selection of a conceptual cohesion metric was the first step of this study. C3 (Conceptual Cohesion of Classes) (Marcus; Poshyvanyk; Ferenc, 2008), CLCOM5 (Conceptual Lack of Cohesion of Methods) (Ujhazi et al., 2010), MWE (Maximal Weighted Entropy) (Liu et al., 2009) and LCbC (Lack of Concern-based Cohesion) (Silva et al., 2012) are the most recent conceptual cohesion metrics available in literature (see also Chapter 2). On one hand, C3, CLCOM5 and MWE directly depend upon text mining techniques to identify concerns in source code and then compute cohesion. On the other hand, for the same purpose, LCbC is not dependent upon a specific technique. LCbC quantifies cohesion of a given module in terms of the number of concerns addressed by it. It just counts the number of concerns identified in each module. Therefore, it is possible to compute LCbC by having any concern mapping strategy applied beforehand. As other conceptual cohesion metrics strongly depend upon a specific concern mapping technique, this study uses LCbC as the representative metric for conceptual cohesion.

Then, the next step was the selection of concern mapping strategies to be applied. We considered the following three: (i) manual mapping based on source code analysis made by developers who do not work in the development of the target system; (ii) XScan (Nguyen et al., 2011) mapping strategy, which is an automatic hybrid approach mixing static analysis using text mining heuristics and structural dependency analysis; and, (iii) Topic-based automatic mapping (Savage et al., 2010) through LDA (Latent Dirichlet Allocation), which is a text mining technique for extracting topics from source code comments and identifiers.

Regarding the target systems, we considered the same from previous studies presented in Chapters 3 and Chapter 5. In order to make the reading of this chapter easier we repeat in the next subsection a brief description about the analyzed systems. If the reader is familiarized with them we suggest skip next section. After generating the concern mappings for all systems under study we then measured LCbC according to each mapping strategy. Then, the next step was the change count measurement for each module of all systems. Section 6.1.4 explains how we carried out the change count measurement in order to address the seventh research question. It is the same procedure presented in the previous chapter where we also counted changes in modules. Thereby, Section 6.1.4 can be skipped if the reader is aware of how we measured changes.

6.1.1 Analyzed Systems

Table 6.1 summarizes the systems we used in this study. Firstly, we selected Rhino 1.6R5¹ and jEdit 4.3.2². Rhino is an open source javascript engine written in Java by the Mozilla Developer Network. It is typically embedded into Java applications to provide scripting to end users, and it is the default Java scripting engine embedded in the official Oracle's Java Development kit. Rhino has been analyzed in different studies recently (Eaddy et al., 2008b) (Dit et al., 2011) (Hill; Rao; Kak, 2012), and Mozilla Developer Network has been maintaining Rhino over the past 15 years involving thousands of commits so far.

jEdit is a long-lived open source text editor for programmers written in java, having thousands of downloads, and highly rated at SourceForge repository. Its commit history is also rich, which reveals an intensive development activity, involving more than 23.600 commits so far. Similar to Rhino, jEdit has been studied in many other works related to software metrics, repository mining and concern mapping (Falleri et al., 2011) (Dit et al., 2011) (Ryssselberghe; Demeyer, 2007). As part of this work, we carried out a manual concern mapping on jEdit code as described in Section 6.1.3. Besides Rhino and jEdit, we did not find any other available system with a concern mapping manually done and having a rich change-set, in order to enable us measuring class change count and addressing the seventh research question.

Additionally, we applied two automatic concern mapping strategies (see Section 6.1.2) to Rhino, jEdit and other four systems in widespread use: Tomcat 6.0.26³, JFreeChart

¹<https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino>

²<http://jedit.org/>

³<http://tomcat.apache.org/>

1.0.6⁴, Findbugs 1.3.5⁵ and Freecol 0.84⁶. Tomcat is one of the most traditional and reliable open source Java web containers available with more than 15 years of active development so far. JFreeChart is another 15 year project and the most widely used open source chart library for Java. Findbugs is a static analysis tool written in Java to find bug patterns in bytecode, and has been used by many major companies such as Google. Freecol is an open source civilization-like game written in Java with more than 12 years of active development and also highly rated at SourceForge.

As the automatic mapping strategies require less effort we could analyze four more systems besides Rhino and jEdit in order to enrich our data-set and make it more heterogeneous. Table 6.1 summarizes the analyzed systems including the release ID, system size in terms of lines of code and also information related to the change-set: repository, number of revisions and months analyzed.

Table 6.1 Systems and change-set analyzed

	Rhino	jEdit	JFreeChart	Findbugs	Tomcat	Freecol
Release	1.6R5 (Nov/2006)	4.3.2 (May/2010)	1.0.6 (June/2007)	1.3.5 (Sep/2008)	6.0.26 (Mar/2010)	0.84 (Aug/2009)
Description	Mozilla's Javascript interpreter	Text editor for programmers	Chart library for the Java platform	Static analysis tool to find bugs in programs	web server container for the Java platform	A civilization-like game
LOC	59,182	109,446	76,059	98,914	161,735	75,902
Repository	Git (Mozilla)	SVN (SourceForge)	SVN (SourceForge)	SVN (SourceForge)	SVN (Apache)	SVN (SourceForge)
# of revisions	2,765	5,851	3,271	4,612	8,941	4,808
# of months analyzed	94	51	86	71	53	60

6.1.2 Automatic Mapping Strategies

XScan Mapping: The XScan (Nguyen et al., 2011) automatic mapping strategy consists of a tool and a mix of static dependency analysis methods and textual analysis of code similarities. It was also applied in our previous studies presented in Chapters 3 and 5 for measuring LCbC. To make this chapter self-contained we briefly repeat here how XScan works.

The goal of XScan is to collect groups of methods that participate together in the realization of a concern. First, XScan searches for pairs of methods as candidates to be part of a concern. Two methods will be part of a concern if they satisfy at least

⁴<http://www.jfree.org/jfreechart/>

⁵<http://findbugs.sourceforge.net/>

⁶<http://www.freecol.org/>

one of the following conditions: (i) they have similar portions of code in their body; (ii) they override or implement the same ancestor method; or (iii) they have similar names. Then, after all possible pair of methods are detected, XScan builds a graph in which nodes represent the methods and edges represent methods relations. Each connected component of that graph might be reported as a concern. Therefore, this is an approach that puts together different techniques which can heuristically suggest groups of methods as concerns realizations mined from the source code.

The XScan solution was proposed recently and the authors reported results indicating more than 90% of accuracy for finding concerns which are addressed by several modules (Nguyen et al., 2011). We made available at the companion website (Silva, 2015b) the complete XScan output with the concern mappings. This output shows a list of method groups each one realizing a concern. Also, at the website it is possible to download our XScan extension for computing LCbC and generating a csv file.

Topic-based Mapping: Also, for Topic-based mapping we used the same tool (TopicXP) and text mining technique (LDA) applied in previous studies explained in Chapters 3 and 5. LDA (Latent Dirichlet Allocation) (Blei; Ng; Jordan, 2003) is a generative probabilistic model for collections of discrete data such as text corpora. In our context LDA is used to automatically discover a set of *topics* within a *corpus*. A corpus is formed by a set of *documents*. Each document is expressed as a probability distribution of topics. Each topic is itself a probability distribution of *words* that co-occur frequently in a corpus of text. Words can belong to multiple topics, while documents can contain multiple topics.

LDA require a set of input parameters for its execution. We used the same parameters explained in 3. For more details about how we executed LDA topic modeling, refer to Section 3.1.2. However, the difference in this study is that we took the Topic-based mapping generated by TopicXP and measured LCbC instead of measuring MWE. Therefore, for each class we counted the number of topics assigned to them. The result of this counting for each class is their corresponding LCbC value.

6.1.3 Manual Mapping Strategy

We applied the strategy of manually mapping concerns to source code, besides the two automatic strategies presented in previous sections. The main reason we used manual mapping is the opportunity to contrast concern mappings carried out by automatic strategies with the ones made by software developers. This would provide a crisp notion about their differences on conceptual cohesion measurement and consequent impact on change-proneness.

For example, one main difference between automatic and manual mappings is that in automatic mappings there is not a predefined list of concerns to be searched by underlying techniques, while in manual mappings developers may have in advance a list of possible concerns extracted from other artifacts. Therefore, concerns identified by automatic mapping strategies may not exactly represent the ones that developers perceive as concerns addressed by the system. Developers may have other information in addition to the source code text.

The manual mappings were based on the Prune Dependency Rule proposed by Eaddy

et al. (2008b), which states that *“a program element is relevant to a concern if it should be removed, or otherwise altered, when the concern is pruned”*. Developers carried out the concern mapping task by systematically inspecting the source and deciding if the prune dependency rule applies to any of the concerns. In some cases, this decision is trivial, e.g., any method named “saveFile” has a prune dependency on the “file saving” concern. The task is highly dependent on how well the developers understand code under analysis. To aid in code comprehension and to decide what concerns to find, developers relied on other project artifacts such as user guides and test cases, and on basic IDE features such as code navigation and search tools.

We applied the manual mapping strategy for Rhino and jEdit. Rhino’s manual mapping was carried out by other researchers and software developers in a different study (Eaddy et al., 2008b), and it has been also used in other works (Dit et al., 2011) (Hill; Rao; Kak, 2012). Rhino is a Javascript interpreter which follows the EcmaScript standard. Therefore, developers who performed Rhino’s manual mapping followed the EcmaScript standard specification. As a result, every normative section of the specification was considered as concerns to be mapped in source code. To persist the association between code elements and the concerns they used the ConcernTagger tool (Eaddy et al., 2008b). This tool stores in a embedded database every mapping relation between a code element (such as methods, fields or the whole class) and a named concern. In addition, we extended ConcernTagger to compute and report LCbC measurement. The companion website (Silva, 2015b) includes links for the Rhino concern mapping and the extended version of ConcernTagger.

For jEdit, we did all the manual concern mapping from scratch. Two researchers⁷, co-authors of this study, carried out this activity which was basically divided in two stages: (i) the initial stage when we invested effort on building an initial set of concerns; and (ii) the mapping stage, when we iteratively and incrementally performed the concern mapping until covering all the set of concerns. The set of concerns is a list of concern names and their description. For example, Table 6.2 shows three concerns in jEdit set of concerns. The complete set is available in Appendix I and on the web (Silva, 2015b).

Table 6.2 A sample of jEdit set of concerns

Concern name	Description
Transferring text	Set of commands and actions for moving and copying text (including cut, paste and copy)
Rectangular Selection	Selection of a text fragment by dragging the mouse with the control key held down for creating a rectangular selection.
Syntax Highlighting	It is the display of programming language tokens using different fonts and colors.

During the mapping stage, we also evolved the initial set of concerns. So, as long as we mapped concerns we also revisited the set of unmapped concerns to think and discuss

⁷Bruno Carreiro da Silva and Neylor Rocha.

about what we were listing as candidate concerns. To build the set of concerns and have more information about jEdit concerns we took as input the following resources:

- The features list available at <http://www.jedit.org/>, which is different from what we mean as “set of concerns”. For instance, “tool bar” is not a feature considered in the features list at jEdit website but it is an important concern implemented by several classes and methods in the project.
- The user guide available at <http://www.jedit.org/users-guide/>.
- The source code itself - jEdit release 4.3.2, provided by the Qualitas.class corpus (Terra et al., 2013).
- System execution. We ran the jEdit 4.3.2 to explore features which had not been previously identified as candidates from analyzing other resources; and also to confirm what we had pointed as candidates.
- Domain knowledge. As programmers, we extensively use a variety of text and code editors. jEdit has several features also found in many other text and code editors we have used.

Overall, the mapping process involved forty seven concerns and two researchers. We split the set of concerns in two parts where each one was assigned to a researcher. This would be unfeasible in terms of time and cost having each researcher performing the entire mapping followed by an additional merging and conflict resolution effort. As a result, the mapping effort of the two researchers totaled 25 hours in several working sessions interspersed with other activities during their working days. Along this process, we studied the user guide many times to confirm or disconfirm candidate concerns or to include new ones in the set of concerns to be mapped. The user guide was very useful to clarify the concepts and behavior behind each jEdit concern. For example, we had doubts about what a switching buffer means in jEdit. By reading the user guide we could have clear explanation about it. In addition, at some points during the mapping stage we executed jEdit to confirm or disconfirm candidate concerns, to include new ones or to have a crisp notion about how a concern is manifested through the graphical user interface.

To support the activity of storing the mapping between source code elements and concerns we also used the ConcernTagger tool. The following steps summarize typical mapping tasks we carried out:

1. Pick a concern from the set of unmapped concerns.
2. Register the time corresponding to the beginning of the mapping task to track the time effort dedicated to map each concern.
3. Define a set of strings and regular expressions to query source code fragments implementing the desired concern. For instance, “encoding; char*enc; enc*char; charset” was the set of strings and regular expressions used for finding candidate code elements to the *character encoding* concern implementation.

4. Run the Eclipse text search using the defined set of strings and regular expressions.
5. Navigate over the results of text-based search, check whether each occurrence in searched results contributes to the concern implementation. If so, assign the code fragment to the concern using ConcernTagger plug-in. As said before, to decide if a code fragment implements a concern, we used the prune dependency rule. If the occurrence corresponds to a statement within a method, we assign the enclosed method, as ConcernTagger does not support concern assignment to statements. Only attributes, methods and entire classes are supported. Additionally, if we identify a method entirely contributing to a concern we then search for other occurrences calling such method, according to the prune dependency rule.
6. If the set of strings and regular expressions defined in step 3 is not entirely covered, go to step 4. Otherwise, proceed to next step.
7. When there is no more code fragments to visit neither search queries to execute, we register the finishing time as well as the strings used for searching concern implementation occurrences.
8. During this activity, in case of any doubt regarding the procedure, source code or any concern, write down a piece of text in corresponding documentation for further discussion in next meeting. This is to make sure that doubts raised during the mapping process could be discussed and double checked.

After every five concerns each researcher mapped we arranged a meeting to: (i) briefly discuss the mapping activity; (ii) solve possible conflicts and doubts; (iii) and align what each one had previously done such as time spent to map the five concerns, and size of each mapped concern in terms of number of classes and methods; and (iv) communicate new concerns discovered during the mapping activities that are candidates to be added to the set of concerns. After covering all the set of unmapped concerns, we arranged a meeting to make a final discussion about the whole process experience and lessons learned. The complete jEdit concern mapping used in this study is available in our online resources page (Silva, 2015b) to be opened with the ConcernTagger plug-in.

6.1.4 Change Count Measurement

Change count measurement is necessary to assess the correlation between change-proneness and respective variations of conceptual cohesion measurement. Therefore this should support us on addressing the seventh research question. The same change count measurement presented in previous study (Chapter 5) was used here, which consists of counting the number of revisions each class undergone along its evolution. For instance, if a class m was changed in 8 revisions throughout the analyzed period of time then $CC(m) = 8$. We considered each commit in the source code repository as a revision. Table 6.1 presents the number of revisions and the corresponding number of months we took into account for each system. We considered the period of time from the selected releases until July 2014 (the month when we completed this study phase).

6.2 RESULTS AND ANALYSIS

This section presents results regarding the application of three concern mapping strategies involving six systems. Over the concern mapping results we analyzed how LCbC varied (Section 6.2.1) and whether this variation impacted on the correlation with change-proneness (Section 6.2.2). With this analysis we could point out several findings described in this section.

6.2.1 Concern Mappings and LCbC Distributions

Table 6.3 shows some numbers related to the application of different concern mapping strategies over the systems under analysis. For each system and each concern mapping we show: the number of mapped concerns; the percentage of Java files with at least one concern mapped; and four properties related to the corresponding LCbC distribution (min and max values, median and standard deviation). For example, in Rhino, XScan mapping identified 26 concerns touching 32% of the Java files. The minimum LCbC over XScan mapping in Rhino was 0, the maximum was 11, the median was 0, and the standard deviation was 2.4. This means that there are classes without concerns mapped by XScan (classes with $LCbC = 0$) and the highest number of concerns that XScan mapped to a class in Rhino was 11 (classes with $LCbC = 11$). The “N/A” cells refer to the systems we did not have a corresponding manual mapping. As explained in Section 6.1, we had the manual mapping strategy only for Rhino and jEdit. In addition, Figures 6.1 and 6.2 illustrates through boxplots the analyzed LCbC distributions. The reasoning guiding the analysis applied in this section is to assess whether and how differences on LCbC distributions could be explained by the applied concern mapping strategies. This supports us on addressing first and second research questions.

6.2.1.1 Auxiliary Artifacts Drive the Accuracy of Manual Mappings As explained in Section 6.1.3 the manual mapping strategy was applied for two of the six systems in this study: Rhino and jEdit. Table 6.3 shows that for Rhino the manual strategy mapped 417 concerns and covered all the Java files, whereas for jEdit it mapped 47 concerns covering 57% of Java files.

The question which arises in this analysis is *why did jEdit, which is bigger than Rhino, has less concerns mapped than Rhino when applied manual mapping strategy?* The reason behind this issue is that manual mappings, as expected, are subjective and mainly depends on two factors: (i) individual reasoning of the developers who carried out the mapping procedure; and (ii) the artifacts they used as input to support them on understanding the system and finding concerns on source code. In our study the two manual mappings varied in these two factors.

For Rhino we used a concern mapping previously reported in literature (Eaddy et al., 2008b), while for jEdit the manual mapping was carried out in the context of this study by two researchers of our group, who have academic and industrial experience on software development in Java. Hereafter, we use the term “developers” referring to those who performed the mapping tasks but did not participated in the development of Rhino

Table 6.3 Summary of Concern Mapping Strategies Applied

Rhino			
	Manual Mapping	XScan	Topic-based
Concerns found	417	26	40
Java files with concern(s)	100%	32%	100%
LCbC min.	17	0	1
LCbC max.	344	11	4
LCbC median	289	0	2
LCbC sd.	130.9	2.4	0.7
jEdit			
	Manual Mapping	XScan	Topic-based
Concerns found	47	97	69
Java files with concern(s)	57%	36%	95%
LCbC min.	0	0	1
LCbC max.	27	11	4
LCbC median	1	0	2
LCbC sd.	2.44	2.4	0.7
JFreeChart			
	Manual Mapping	XScan	Topic-based
Concerns found	N/A	282	70
Java files with concern(s)	N/A	46%	93%
LCbC min.	N/A	0	0
LCbC max.	N/A	37	6
LCbC median	N/A	0	2
LCbC sd.	N/A	3.5	1
Findbugs			
	Manual Mapping	XScan	Topic-based
Concerns found	N/A	151	91
Java files with concern(s)	N/A	32%	92%
LCbC min.	N/A	0	0
LCbC max.	N/A	10	6
LCbC median	N/A	0	2
LCbC sd.	N/A	1.6	0.9
Tomcat			
	Manual Mapping	XScan	Topic-based
Concerns found	N/A	319	92
Java files with concern(s)	N/A	35%	96%
LCbC min.	N/A	0	0
LCbC max.	N/A	20	13
LCbC median	N/A	0	2
LCbC sd.	N/A	2.4	1.9
Freecol			
	Manual Mapping	XScan	Topic-based
Concerns found	N/A	91	66
Java files with concern(s)	N/A	56%	98%
LCbC min.	N/A	0	0
LCbC max.	N/A	13	4
LCbC median	N/A	1	2
LCbC sd.	N/A	2.1	0.8

and jEdit.

Intuitively, concern mappings manually performed by different developers are expected to be different even if they have the same intent, as they apply distinct and individual reasoning. Additionally, developers used different artifacts to support them on understanding source code during mapping tasks. As Rhino is a Javascript engine based on the EcmaScript standard the developers used the standard specification as input, which is a detailed document describing how the scripting engine should behave. Thus, they considered each section and subsection of the standard specification document as a single concern to be addressed and manifested in code. Also, to help them in finding which code elements implement each section of the document they used unit tests available in the repository. Each unit test is dedicated to test the adherence of Rhino to the EcmaScript standard.

For jEdit there is not such standard specification, so we mainly followed: the user guide; the features list published on the web; and system execution for searching concerns on the graphical user interface and for understanding how some concerns behave externally, as explained in Section 6.1.3. Therefore, in jEdit the gap between input artifacts for concern mapping tasks and source code is bigger. So it turns out to be more difficult to cover a high number of software concerns and files accurately as in Rhino. Figure 6.1 shows boxplots for LCbC distributions using manual mapping, including minimum and maximum values, median and standard deviation. We use the “LCbC_MM” label to represent LCbC measurement with manual mapping strategy. The boxplots show that, in Rhino’s distribution, besides having more concerns it also has more variance. jEdit’s mapping led to a less spread LCbC distribution. Most of the classes have 0 to 4 concerns, whereas only 3% of them have 5 or more concerns.

In summary, developers who carried out Rhino’s mapping had a more precise documentation in hand, which was actually the same one used by Rhino’s developers for coding, testing and addressing EcmaScript standard. This is the main reason in Rhino there is a more detailed mapping with more than 400 concerns touching all the system files. Therefore, which auxiliary artifacts developers use for manual mapping are crucial for determining mapping accuracy, and this certainly influences on conceptual cohesion distribution.

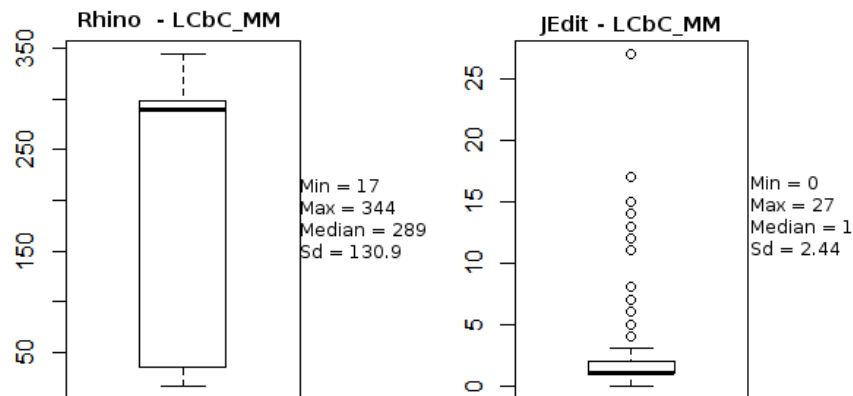


Figure 6.1 LCbC distributions using manual mapping strategy

6.2.1.2 Less Coverage of XScan Mappings Turning to the other mapping strategies, they are automatically executed and we used them for all systems. As explained in Section 6.1.2, XScan strategy only needs the source code as input. Its own heuristics are dedicated to find groups of correlated methods mainly based on structural dependency and textual similarity of statements and identifiers. Each group of methods in the XScan output was treated as a concern.

File coverage on XScan mappings varied from 32% to 56% (see Table 6.3), which represents the least coverage over project files among the mapping strategies we analyzed. XScan does not favor a high coverage over files because it was not designed for it. In Section 6.2.1.5 we highlight some characteristics of XScan strategy after analyzing mapping outputs. In summary, there are two main reasons for such low coverage. First, not all concerns manifest over methods with statements or identifiers similarly defined and over methods structurally dependent with each other. Second, XScan was originally designed for finding concerns spread over several files without having file coverage as a primary issue. Therefore, concerns addressed by one or few files may not be captured by XScan. This may lead to classes without concerns mapped to them. Moreover, the LCbC distributions using XScan had median zero in all the systems except for Freecol with median 1.

6.2.1.3 Low LCbC Variance of Topic-based Mappings For Topic-based mapping, the number of topics is one of the parameters needed. We calculated it based on the number of Java files (see Section 6.1.2). Therefore, the number of concerns is exactly the calculated number of topics as each topic is treated as a concern. Different from XScan, this mapping strategy leads to a high coverage over Java files, as it varied from 92% to 100%.

Figure 6.2 presents boxplots of LCbC distributions using XScan and Topic-based mappings. It graphically shows that Topic-based mappings lead to slightly higher medians in LCbC distributions but less variance in comparison with XScan mappings. Therefore, although Topic-based mappings cover almost all the system files, these mappings lead to flatter LCbC distributions. In other words, the Topic-based strategy maps almost the same number of concerns to each Java classes. The standard deviation, min and max values in Table 6.3 also give evidence to this observation.

In summary, manual mappings generate LCbC distributions with more variance, whereas automatic mappings lead to flatter LCbC distributions. Comparing the two automatic strategies, on one hand XScan generates mappings with low to medium file coverage and provides LCbC distributions with low median but higher variance. On the other hand, Topic-based mappings cover more files and lead to LCbC distributions with slightly higher medians but lower variance. In other words, XScan leaves many classes without concerns mapped but it is able to identify classes focusing on several concerns, whereas Topic-based strategy leaves less classes without concerns mapped but all classes end up with an almost uniform number of concerns. This analysis shows that Topic-based strategy leads to less realistic mappings. In real world projects, like the ones in this study, there are classes realizing several concerns, while others are focused on one or just few concerns.

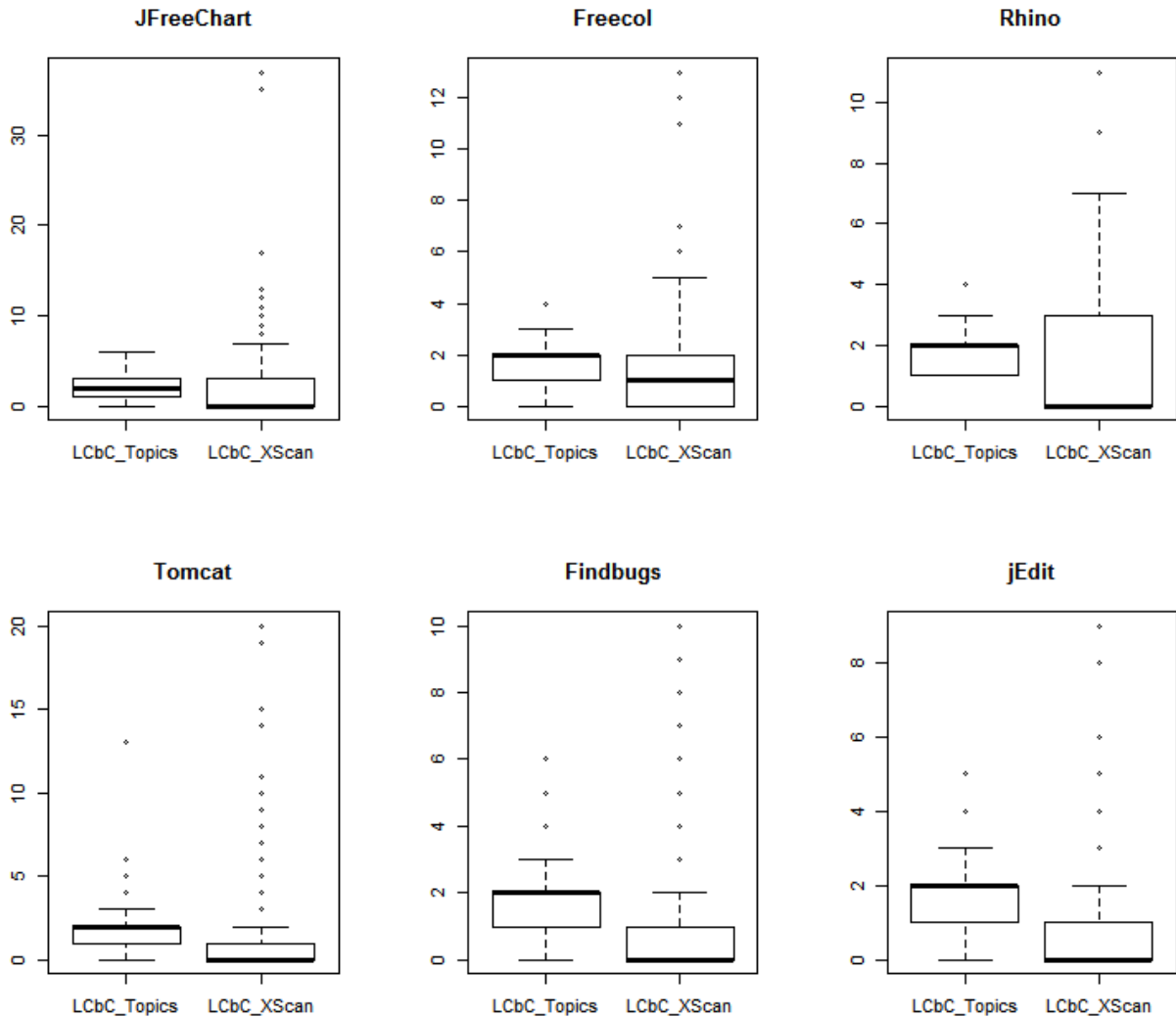


Figure 6.2 LCbC distributions using XScan and Topic-based mapping strategies

Overall, this comparative analysis gives empirical evidence that as we vary the concern mapping strategy, we also have a significant variation on LCbC measurement. Thus, different concern mapping strategies imply different LCbC distributions over the same set of classes.

6.2.1.4 Statistical Tests We applied the Friedman test (Neave, 2010) for LCbC measurements of each system under analysis. Friedman test is a non-parametric test which is used to detect differences in treatments across multiple measurements. In this study, for Rhino and jEdit we have three variations of LCbC measurement, whereas for JFreeChart, Findbugs, Tomcat and Freecol we have two variations of LCbC. Therefore, we set the following null hypothesis: *given a system, the LCbC distributions, each one corresponding to a concern mapping strategy, are the same.* After applying the Friedman test in R over the LCbC distributions we rejected the null hypothesis for all systems with

a p-value less than 0.00. Thus, for all the systems and concern mapping strategies under study we could statistically validate the hypothesis that different mapping strategies indeed drive different distributions of conceptual cohesion. The complete script and log for this test execution in R are available at Appendix H and on the web at (Silva, 2015b).

6.2.1.5 Characterizing XScan Mapping Results To have a concrete notion about what the automatic mappings generate as results and understand what kind of concerns they identify, we analyzed the outputs of XScan and TopicXP tools. In XScan, for each system, we analyzed the output files containing groups of methods. In TopicXP, for each system, we analyzed the list of words forming each identified topic. We considered each group of methods generated by XScan and each topic generated by TopicXP as a concern. The output files for all systems are available at our companion website (Silva, 2015b).

XScan mapping identifies concerns only manifested in methods by using code similarity analysis and structural dependency. We found that it typically captures concerns implemented over different classes which uses the same method name. Concerns mapped by XScan are not labeled. However, in general, it is relatively easy to identify what the concerns mean, by reading method signatures and class names forming each concern.

There are situations where XScan goes right on the target, for instance, the *stop* concern in Tomcat. It implements the stopping mechanism of the web container. In source code it is addressed by several implementations of the *stop()* method. This concern involves 39 classes from 11 packages. XScan was able to identify this concern mainly because the involved classes use the same method name. If there are other methods involved in this concern but with other names, XScan is not able to identify them.

In some situations XScan heuristics identify very specific concerns like the “dialog canceling action” for window dialogs in jEdit system. This concern involves 20 different classes that implement the dialog canceling action through the *cancel()* method.

However, XScan fails to identify concerns that do not manifest themselves through similar method names. For instance, the JEditBuffer class in jEdit. This class had no concerns mapped by XScan, while it is an important class that represents the contents of an open text file as it is maintained in computer memory. When looking at the class we clearly find concerns like load/unloading content, inserting/removing text into the buffer and text indentation.

These examples reinforce observations over Table 6.3 and Section 6.2.1.2 that XScan strategy leads to less coverage over classes. However, in spite of such characteristic XScan is able to identify some classes with several concerns. Table 6.3 shows that XScan mapping leads to higher max values for LCbC than Topic-based mapping. We can also observe some high values of LCbC_XScan in the boxplots (Figure 6.2) when compared to LCbC_Topics. Those XScan characteristics lead to an incomplete mapping in general, but it is able to capture some critical classes in terms of lacking conceptual cohesion.

6.2.1.6 Characterizing Topic-based Mapping Results Topic-based mapping identifies concerns manifested through correlated words within classes, including identifiers (at class, field and method level) and comments embedded in source code. The topics

extracted are not labeled. We manually analyzed the topics by looking at the words that form them. We observe similar results also reported previously in literature (Lucia et al., 2012): while some topics are easy to label and give a notion about what they mean, some other topics are difficult to label, as their words are not meaningful enough.

For instance, when reading the topic [plugin, jar, path, name, file, manag] we could label it as the *Plug-in manager* concern, which is responsible for managing plug-ins as jar files into jEdit editor. However, some topics are hard to understand. For instance, the words in the following topic are meaningful [jedit, edit, method, j, i, param] if you analyze them separately. However, when considering them together forming the topic they are meaningless, becoming difficult to put a label on it in jEdit context. Even with the previous experience of carrying out a manual mapping over jEdit code we could not find a label to this topic.

In addition, after analyzing the classes having that topic, we could observe that “jedit” and “edit” words are very common in the project and it appears as class and variable names in many places. Particularly, the “method” word appears in comments attached to several methods across the project. The words “i” and “j” are common in classes having loops. “Param” is a common word appearing in javadoc comments withing several classes. In fact, these words are examples of textual noise for the Topic-based mapping strategy. They should have been ignored by the LDA implementation in TopicXP tool, as they do not add meaning to a relevant topic in the overall system. This leads us to conclude that in order to improve the results the Topic-based mapping strategy requires additional effort to filter out general and meaningless words from the source code. Ideally, in addition to the list of English stop words as part of the pre-processing step of LDA execution, there should be a list of project-specific stop words to be used in LDA applications over source code such as Topic-based concern mapping.

6.2.1.7 XScan Requires less Effort It is obvious that manual mapping requires more effort and so it is more expensive to apply. jEdit manual mapping took 22 hours of work, while Rhino took 102 hours, as reported in (Eaddy et al., 2008b). Turning to the automatic strategies, XScan is easier to execute than Topic-based strategy for two reasons: (i) besides the source code, it does not require any input parameters, neither effort to parameters calibration; and (ii) it takes less time to process and generate results. For instance, TopicXP took 6 hours to generate the results for Tomcat, which is the biggest system we analyzed, while it took less than an hour using XScan in the same computer.

6.2.1.8 They are different but are they correlated with each other? We found that LCbC distributions have different characteristics when varying the concern mapping strategy for the same set of classes. However, the question which arises is: *being different, are they correlated?* In other words: *Given a system, are the different LCbC distributions correlated with each other?* If two different LCbC distributions are correlated it is an evidence that we can choose one of them in future studies in order to reduce effort of generating both of them. We applied Spearman correlation (Cohen, 1988) to address this question. Table 6.4 shows the correlation test results for Rhino and jEdit with

“*” representing no significance level achieved. In addition, Table 6.5 shows the same correlation test applied for the other four systems which we have only the automatic mapping strategies.

In order to give a qualitative label for the obtained correlation coefficients, we follow Cohen’s range (Cohen, 1988). Such a range suggests that a coefficient from .00 to .30 is a weak correlation; from .30 to .50 is a moderate correlation; and higher than .50 is a strong correlation. For Rhino, we found moderate correlation between LCbC over manual mapping (LCbC_MM) and LCbC over XScan mapping (LCbC_XScan). And also moderate correlation between LCbC_XScan and LCbC over Topic-based mapping (LCbC_Topics). For jEdit, we found a weak correlation among the three LCbC measurements.

Table 6.4 Correlation among LCbC variations for Rhino and jEdit

Rhino			
	LCbC_MM	LCbC_XScan	LCbC_Topics
LCbC_MM	1	0.38	*
LCbC_XScan		1	0.39
LCbC_Topics			1
jEdit			
	LCbC_MM	LCbC_XScan	LCbC_Topics
LCbC_MM	1	0.27	0.22
LCbC_XScan		1	0.16
LCbC_Topics			1

Turning to the other four systems, the correlation analysis consists of comparing only LCbC_XScan with LCbC_Topics. For this set, the results also varied with significance level achieved. We found almost no correlation between LCbC_XScan and LCbC_Topics for Tomcat (0.08 coefficient), whereas this correlation varied from weak (Findbugs and Freecol) to moderate (JFreeChart). Therefore, we did not find any strong correlation among LCbC distributions. This is another evidence that variation on concern mapping strategy significantly impacts on conceptual cohesion measurement. Also, this correlation analysis, together with previous observations in this work, supports us not in recommending the replacement of one strategy over another for measuring conceptual cohesion, unless having a specific reason.

Table 6.5 Correlation among LCbC variations for JFreeChart, Findbugs, Tomcat and Freecol

LCbC_XScan vs. LCbC_Topics	
JFreeChart	0.41
Findbugs	0.28
Tomcat	0.08
Freecol	0.29

In summary, according to our results, there is a significant impact on conceptual cohesion measurement when varying the concern mapping strategy. Manual mappings can drive different LCbC distributions depending on what auxiliary artifacts developers have to carry out mappings. Comparing to Topic-based strategy, XScan executes faster but

has less coverage over Java classes, leaving more classes without concerns mapped. However, XScan is able to identify classes with several concerns, and it scores on mapping any kind of concern manifested through similar method names. In counterpart, XScan fails when concerns are not expressed like that. Topic-based strategy captures well concerns expressed in identifiers and comments, however it suffers from textual noise. Also, corresponding LCbC distributions have lower variance than XScan ones, which leads to a less realistic mapping. In real software projects there is high imbalance regarding concerns implementation among project classes, that is, while some classes realize several concerns, some others realize just one or few of them. This increases the variance of conceptual cohesion distribution.

Additionally, we need to know whether such different mapping strategies affect the correlation between conceptual cohesion and change-proneness, which is the point of RQ3 and the main issue analyzed in the next section.

6.2.2 Correlation between LCbC and Change-Proneness

As explained in Section 6.1.4 we measured change count (CC) for each class of all systems over a period of time in their evolution history. A summary of the analyzed change-set for each system can be found in Table 6.1. The CC metric is our proxy for change-proneness, having the following interpretation: the higher the CC value for a class, the more change-prone the class is, and vice-versa.

Table 6.6 shows the results after applying Spearman correlation between LCbC of each mapping strategy and change count for all systems. We highlight in bold the highest correlation coefficients of each system. ‘*’ means no significance level achieved. Considering the two systems we had manual mappings, LCbC_MM had the highest coefficients, achieving the strong range in Rhino and moderate in jEdit. Considering all projects, LCbC_XScan and LCbC_Topics had similar coefficients. However, LCbC_XScan achieved a strong correlation in JFreeChart, whereas LCbC_Topics achieved moderate correlation in the best case.

Table 6.6 Correlation between Change Count and LCbC measurements

	Rhino	jEdit	JFreeChart	Findbugs	Tomcat	Freecol
LCbC_MM	0.67	0.38	N/A	N/A	N/A	N/A
LCbC_XScan	0.32	0.18	0.63	0.21	0.33	0.47
LCbC_Topics	*	0.27	0.41	0.30	0.26	0.45

6.2.2.1 Mapping Size Does not Matter Taking into account the LCbC distributions graphically shown in Figures 6.1 and 6.2 and also descriptive statistics in Table 6.3, we could not find any possible association between change count and mapping size in terms of number of concerns and class coverage. For example, in jEdit, Topic-based mapping covered more classes than the other strategies, while XScan found more concerns. However, LCbC over manual mapping had the highest correlation with change-proneness. A high number of identified concerns or covered classes do not necessarily mean a complete and accurate mapping neither increase the correlation with change count. Therefore, we

can conclude that concern mapping size does not necessarily impact on the correlation between conceptual cohesion and change-proneness.

6.2.2.2 XScan Mapping when Manual Mapping is not Possible When compared to automatic mapping strategies, manual mappings had the highest correlation coefficients, achieving a strong correlation in Rhino. The mapping quality is more important than number of concerns and number of files covered. In other words, the main driver for better mappings is how well the concern assignment to source code elements can represent what the software performs. In general, manual mappings are at better quality as developers, in theory, extract more accurate knowledge from analyzing code than automatic techniques currently available. When developers code following a specification such as requirements, use cases, or a standard as in Rhino, they address in source code the concerns present in such higher level artifacts. Therefore, when such artifacts are also used as input for manual concern mapping this enables building a better outcome. We found evidence that more complete concern mappings, such those found in Rhino and jEdit manual mappings, increase the correlation between conceptual cohesion and change-proneness.

When manual mapping is not possible to be applied we recommend the XScan mapping strategy. In terms of automatic mappings, LCbC with XScan achieved a strong correlation in JFreeChart, showing that it is also possible to have automatically generated concern mappings with moderate to strong correlation between conceptual cohesion and change-proneness. In addition, as discussed in Section 6.2.1, when compared with Topic-based mappings, XScan: (i) requires less effort, because it executes faster and does not require calibration of input parameters; (ii) is able to identify classes with several concerns which may be critical classes in terms of lacking conceptual cohesion.

In summary, these results confirm that varying concern mapping strategies impacts on conceptual cohesion distribution and therefore such variation significantly impacts on the correlation between conceptual cohesion and change-proneness. Manual mappings seem to reflect better what the modules address and so improve such correlation. In general, manual and XScan strategies showed more potential to drive higher correlations between conceptual cohesion and change-proneness.

6.3 THREATS TO VALIDITY

Internal validity: This type of threat may have a negative effect on the occasional relationship between treatment and outcome. In this study we identified two possible threats to internal validity. First, it is about how we addressed RQ5 and RQ6. We mitigated possible occasional relationship between concern mapping strategies and conceptual cohesion by providing empirical evidence through qualitative and quantitative analysis. We qualitative analyzed how and in which context the mappings were built and what are the main characteristics of mapping results. We also used descriptive statistics and graphical analysis over LCbC distributions. The Friedman test was then applied just to confirm and add statistically significant support to our observations. Therefore, we claim that the outcome we found between the impact of different concern mapping

strategies and conceptual cohesion measurement is not occasional.

The second threat to internal validity is regarding the correlation between LCbC distributions and change-proneness, that is, whether low/high conceptual cohesion really influenced on more/less class changes (seventh research question). This threat does not become critical in this study as our focus is on a relative comparison between different ways of measuring LCbC and change-proneness. In other words, given a system, what varied was only the concern mapping strategy and that was the desired point of interest in this study. Other factors that could have affected a correlation test would have affected in the same way all the correlation tests of a given system.

External validity: Two factors affect the external validity. First, our study involved three different concern mapping strategies, having the manual strategy applied only in two of the six systems under study. It is very difficult to find an available tool to map concerns which is ready to be used in software projects. We are only aware of XScan and TopicXP tools. Others are prototypes cited in papers or addressing projects in other languages than Java.

The Manual Mapping strategy was considered for only two systems as we did not find any other available project with a concern mapping manually done with the prerequisites we needed for this study. The prerequisites were: (i) a manual mapping intended to cover as much as possible the source code modules and the system concerns of any kind; (ii) implemented in Java, as the tools we used only supports Java systems; (iii) a system with a rich change set to allow us addressing the seventh research question.

The second factor affecting external validity is that we analyzed six open source systems. We believe this number of systems is reasonable to draw the first conclusions in this kind of empirical assessment which were partially done manually. Also, the systems are from distinct domains and also varying from libraries to standalone applications. They are popular and widely used, as discussed in Section 6.1.

Therefore, we do not claim that our conclusions can be generalized outside the scope of our study. However, as far as we know, this is the first study on analyzing and improving knowledge about the impact of different concern mapping strategies on conceptual cohesion measurement. Even restricted to the study settings our findings give evidences to support software engineers on cohesion measurement activities and researchers in their further studies involving conceptual cohesion.

Construct validity: When evaluating the correlation between conceptual cohesion and change-proneness we used the CC (Change Count) metric. We considered this metric as a proxy for change-proneness according to the reasoning that a change-prone class is a module that has high likelihood to undergo changes. Thus, as we count the number of changes we are also counting the change-proneness degree. However, there is a possible threat in this context as we did not analyzed which type of changes each class suffered. We just counted changes without filtering out non-essential changes such as moving folders, license changing and source code comments. We considered every add, delete and modify operations as changes regardless whatever motivated them. So, we took into account changes of any type such as bug fixing, inclusion of new features, preventive maintenance or any kind of refactoring. The analyzed systems and corresponding change history are the same used in previous study (Chapter 5). Therefore, the design decisions to overcome

this threat are also the same. We adopted two requirements on selecting the six analyzed systems: (i) we searched the systems website to find intermediary releases reporting a reasonable number of changes in their later releases; and (ii) a candidate system release should have a repository with a large change scenario ahead in their commit history. Table 6.1 shows that we analyzed commits along 51 months (jEdit) to 94 (Rhino). Therefore, even in the presence of noise the change sets were predominantly composed by essential changes.

The parametrization of LDA execution for Topic-based mappings was identified as another possible threat to construct validity. The calibration of LDA parameters is important to better explore the LDA algorithm and achieve more confident results in Topic-based mappings. However, it is an empirical time-consuming activity which is out of scope of this study, as explained in Section 6.1. To compensate this threat we followed recent literature regarding the application of LDA in source code topics extraction. For the number of topics, we used the equation proposed by (Grant; Cordy; Skillicorn, 2012). For the α and β parameters we followed recommendations from (Binkley et al., 2014).

Conclusion validity: Threats to conclusion validity are lack of statistical calculations or misuse of statistical assumptions that leads to incorrect conclusions made by the researcher. Due to the nature of our first and second research questions, we could not address them only using statistical calculations. Therefore, most of our empirical assessment for RQ5 and RQ6 relied on qualitative analysis, where we analyzed and explained characteristics of concern mapping outputs and how they were generated. However, whenever applicable we used statistical support. In Section 6.2.1 we used descriptive statistics and Friedman test to support us on explaining the impact of mapping strategies on LCbC measurement. The *Friedman test* was applied in place of *Repeated Measures Analysis of Variance* (rANOVA) because it is a non-parametric test. The LCbC distributions do not match the necessary assumptions to apply rANOVA, as the normality assumption for example. For the seventh question we applied Spearman correlation test to support us on explaining whether and to what extent such impact affects the correlation between conceptual cohesion and change-proneness (Section 6.2.2). Also, we chose the Spearman test due to characteristics of LCbC distributions, as it is a non-parametric correlation test. Moreover, Spearman's method can find correlation between variables even if their relationship is not linear. Pearson correlation test would be an alternative. However, it is by definition restricted for finding linear relationship and it is not robust enough if the analyzed distributions have outliers.

6.4 RELATED WORK

The problem of mapping concerns to source code elements is not new (Biggerstaff; Mitbander; Webster, 1994). However, we are not aware of any study that analyzed the impact of different concern mapping strategies on conceptual cohesion measurement. Therefore, in this section we focus on citing studies that investigated the activity of concern mapping and somehow relate to our work.

Eaddy et al. (Eaddy et al., 2008a) proposed a tool called Cerberus which implemented an automated concern mapping strategy combining three techniques: text min-

ing, execution tracing, and structural dependency analysis. They applied each technique individually and combined in pairs over Rhino source code. Also, Dit et al. (Dit; Revelle; Poshyvanyk, 2013) studied the combination of different techniques for automatically mapping concerns. Overall, the results of those two studies pointed out that combination of techniques are more effective for mapping concerns. They did not study the impact on code measurement, and their tools are not available. However, their results match with ours in the way that XScan implements a hybrid approach combining two different techniques (textual similarity and dependency analysis) and that we found XScan outperforming Topic-based strategy, which only implements text mining.

Figueiredo et al. (Figueiredo et al., 2011) assessed the impact of manual concern mappings on code measurement. They analyzed the accuracy of concern mappings individually made by 80 developers and their impact on 12 metrics focused on measuring concern properties. They did not include metrics for software module properties such as cohesion. Also, they did not apply automatic concern mapping strategies. However, one of their results match with ours. They found that developers are usually conservative when mapping concerns, thereby not assigning concerns to elements that they are uncertain about. This somehow relates with our experience on building jEdit manual concern mapping. As we had less auxiliary artifacts than in Rhino's manual mapping, we probably were more conservative on assigning concerns to jEdit code elements. To some extent this may explain why in jEdit, which is a bigger system than Rhino, we had fewer concerns and lower coverage when comparing their manual mappings. We discussed this issue in Section 6.2.1.1.

6.5 SUMMARY

Cohesion measurement is challenging and several researchers have attempted to address this issue across decades. Most of them rely on conventional cohesion metrics which are based on the structural relatedness among methods and attributes within a class. However, this approach does not necessarily capture the degree to which a class is focused on a single software concern.

With the emergence of concern mapping techniques, researchers have investigated cohesion metrics based on abstract information extracted from source code. The Lack of Concern-based Cohesion (LCbC) is a simple metric that counts the number of concerns a software module realizes, and it is flexible to be used over concern mappings resulted from applying any concern mapping strategy.

However, little is known about the impact of different concern mapping strategies on conceptual cohesion measurement. This is the first empirical study to address this issue. This work presented an empirical assessment which improves knowledge about how different concern mapping strategies impact on conceptual cohesion measurement.

We used three different concern mapping strategies: the manual strategy based on developers analysis on source code and other auxiliary artifacts such as user guide and test cases; and two automatic strategies, one named XScan, which is based on code similarity and structural dependency, and another named Topic-based strategy, which applies text mining. In addition, the study involved six popular long-lived open source systems from

different domains.

We found that XScan requires less effort than Topic-based strategy. And the manual mapping is the most time-consuming one and the most expensive to perform. When performing manual mapping, auxiliary artifacts besides source code are crucial for generating accurate results. Although XScan leaves many classes without concerns mapped it is able to identify classes that stand out for having several concerns. Differently, Topic-based strategy leads to quite uniform LCbC distributions, which means that classes in a given project have almost the same number of concerns.

In addition, Topic-based mappings suffer from textual noise, that is, meaningless words for representing concerns used in source code identifiers and comments. Therefore, we recommend to build a list of project-specific stop words to be pre-processed together with English stop words when executing Topic-based strategy.

Overall, variation in concern mapping strategy significantly impacts on conceptual cohesion measurement. The automatic strategies we analyzed are not good substitutes for manual mapping, at least for cohesion measurement and its association with change-proneness. Mapping size does not matter. Mapping accuracy is more important and how the strategy is able to capture critical classes lacking cohesion.

Therefore our results can be used by software engineers when planning and executing cohesion measurement in their activities and when building metric tools for development environments; it can also support researchers on further investigations involving cohesion measurement. We suggest three directions for future work. First, the evaluation of other concern mapping strategies using different techniques such as history-based concern mapping and dynamic tracing source code execution. Second, besides counting changes and correlating them with cohesion, we recommend investigation on analyzing whether changes within modules are associated with the concerns mapped by different strategies. Third, as topic-based mappings were very different from manual mappings, it is also important to investigate whether other cohesion metrics that use topic-based strategies measure cohesion as perceived by developers.

CONCLUSION

Cohesion has been considered throughout decades as one of the most important software quality attributes, and several approaches for measuring cohesion and investigating its application in different software engineering contexts has been produced (Stevens; Myers; Constantine, 1974), (Myers, 1978), (Chidamber; Kemerer, 1994), (Henderson-sellers, 1996), (Briand et al., 2000), (Martin, 2003), (Marcus; Poshyvanyk; Ferenc, 2008). Amongst a number of definitions, module cohesion can be viewed as the degree to which a module is focused on a single concern of the software. Highly cohesive modules are claimed to be easier to understand and evolve than less cohesive modules (Briand; Bunse; Daly, 2001; Chen et al., 2002; Dallal; Briand, 2012).

Building and maintaining cohesive software modules is a desired practice and a number of metrics have been proposed for measuring module cohesion. Most of them are structural cohesion metrics, i.e., they measure the cohesion of a module based on the degree of relatedness of its internal elements. However, structural cohesion metrics expose limitations as they are tightly dependent upon the source code structure no matter the amount of different concerns a module implements. In this context, researchers have proposed an alternative way of cohesion measurement based on which concerns software modules address.

Conventional structural cohesion metrics such as the ones presented in Chapter 2 have been in scene through decades. They are vastly applied on source code measurement as part of empirical studies that include cohesion as a design property among others such as coupling and size. Also, they are present in a number of open source and commercial software measurement tools. However, the emergent set of conceptual cohesion metrics lacks empirical evidence to better understand how they fit among several conventional structural metrics for cohesion measurement. Therefore, this scenario motivated us to conduct a series of empirical studies to fill this gap.

As a result, we could statistically validate conceptual cohesion, represented by two distinct metrics, as an orthogonal way of cohesion measurement in comparison with structural cohesion metrics. This means that a different dimension of cohesion measurement

is captured by conceptual cohesion metrics due to different source of information and the counting mechanism for computing cohesion. The study involved more than three thousand modules of six long-lived, industry-scale, medium to large-sized systems.

Also, we could demonstrate, through an additional study, how developers from different countries and levels of experience perceive module cohesion. Our results supported the claim that conceptual cohesion seems to be more intuitive and closer to the human-oriented view of cohesion observed on software developers. Structural cohesion measurement has been the most common way of measuring cohesion both in academic works and in industry-scale software tools. However, our findings point out that conceptual cohesion measurement captures better the real notion of cohesion as perceived by developers in contrast with the conventional structural cohesion measurement.

We could also contribute on studying the association of conceptual and structural cohesion metrics with module change-proneness. We found that conceptual cohesion is a potential change-proneness indicator when compared to conventional structural metrics, and its success on this role mainly depends upon how well concerns are mapped over the source code.

Therefore, we also studied the impact of different concern mapping strategies on conceptual cohesion measurement. With this later study we could find that when measuring conceptual cohesion, the selection and execution of a mapping strategy should not be neglected as it significantly impacts the measurement results. Among the strategies we studied, manual mapping leads to better results. However, as it requires much effort, XScan mapping should be also considered when the manual strategy is not possible.

All of our study datasets and materials are public available at our companion websites (Silva, 2015a) (Silva, 2015b) (Silva, 2013).

In summary, based on a series of empirical studies, our work explained conceptual cohesion as an additional way of cohesion measurement that is worth to be further considered in software measurement research and practice. Our results can be directly used by software engineers when planning and executing cohesion measurement in their tasks and when building metric tools for development environments. Overall, this justifies additional effort to keep improving cohesion measurement knowledge and technology. Thereby, in the next section we describe possible directions for future research.

7.1 FUTURE RESEARCH DIRECTIONS

We suggest the following directions for future work that arise from our research:

- ***Assessment of additional concern mapping strategies.*** As explained in Chapter 2, there are several techniques that have been proposed as concern mapping strategies. We applied three different strategies in the context of our research. Therefore, we recommend broadening this scope to assess additional concern mapping strategies such as the ones based on history analysis and dynamic tracing source code execution. The former relies on extracting information about what concerns system modules address by analyzing how they change over time. This kind of strategy builds a concern mapping by identifying changing patterns and associating co-changing modules and its internal elements. The latter relies on cre-

ating a concern mapping by processing different execution scenarios of the software under analysis. Thus, this strategy dynamically traces source code fragments involved in each execution scenario in order to capture which methods and attributes are used by each concern. Therefore, a possible study goal could be the investigation of how such additional strategies fit within the two automatic ones we used in our studies with respect to conceptual cohesion measurement. Also, we suggest investigation on whether those additional strategies could be used when manual mapping is not feasible to be applied.

- ***Conceptual cohesion measurement for a specific domain.*** In this current scope we did not select a specific application domain. The studies presented in chapters 3, 5 and 6 involved six long-lived, industry-scale, medium to large-sized systems from different domains such as games, source code analysis tools, language interpreters and text editors. Therefore, after the results we achieved, a future study could focus on whether or not conceptual cohesion measurement would be more effective on a specific application domain.
- ***Deep analysis on whether and to what extent changes within modules are associated to concerns mapped to them.*** We quantitatively assessed the association between several cohesion metrics and module change-proneness. One of our threats to validity was that we did not analyze details about the change context. For example, we did not analyze the kind of changes and what had motivated them. Therefore, we studied the change-proneness issue in a more quantitative way by selecting rich and long change-set periods so that possible noise in this kind of data could be minimized. For that reason there is still room for an additional study on investigating what has motivated changes over modules along their history and whether or not this could be associated to cohesion metrics. We hypothesize that this could reinforce conceptual cohesion metrics as change-proneness indicators by showing that relevant changes suffered by modules are associated to their concerns.
- ***Assessment of whether other conceptual cohesion metrics associate to developers' perception about module cohesion.*** In our study described in Chapter 4, besides understanding developers' reasoning about module cohesion, we assessed how close developer ratings are to conceptual cohesion measurement, having LCbC as a representative. Therefore, we do not know whether topic-based conceptual cohesion metrics, such as MWE and C3, also associate with how developers reason about module cohesion as we found with LCbC.
- ***Broadening scope to systems in other languages.*** Our set of empirical studies and findings relied on projects written in Java. It is reasonable to think that most of our results could also be found in similar widely used object-oriented languages (e.g. C#), as these languages provide object-oriented concepts in a similar way. However, there are also widely used languages, such as Javascript, that have distinct characteristics (e.g. dynamic typing, script-based) besides providing object-oriented programming concepts. A recent study (Silva et al., 2015) showed

that most of the Javascript modules do not use the abstraction of classes. Therefore, we claim that it is necessary to understand and demonstrate how we can measure cohesion of JavaScript modules, specially those which do not apply object-oriented concepts. For those, conventional structural cohesion metrics could not be used. Conceptual cohesion metrics might be a good choice for this purpose.

- ***Assessment of how conceptual cohesion associate to other quality attributes.*** We analyzed the association of conceptual cohesion with module-change proneness and compared how structural cohesion performed for the same association. The main motivation for choosing change-proneness was the fact that cohesion is defined to be a natural change-proneness indicator. Considering that cohesion is defined to be the degree to which a module is focused on a single concern, the more concerns a module addresses the more reasons there are to change it (Martin, 2003). However, it is also important to understand whether conceptual cohesion metrics associate to other quality attributes, such as fault-proneness, comprehension effort and maintenance effort. Recent works (Marcus; Poshyvanyk; Ferenc, 2008) (Liu et al., 2009) have studied the association of fault-proneness with separate conceptual cohesion metrics without comparing them. Recently our research group has started to study the relationship of structural and conceptual cohesion metrics with program comprehension effort (Batista; Sant'Anna, 2015).
- ***Incorporation of conceptual cohesion metrics on measurement tools.*** One of the main outcomes of this current research is that we could provide empirical evidence through a series of empirical studies that conceptual cohesion measurement is promising and it is worth further effort to keep improving cohesion measurement knowledge and technology. Conceptual cohesion can increase its value by taking advantage of concern mapping techniques which suffer continued improvement. Therefore, we recommend the incorporation of conceptual cohesion metrics together with concern mapping techniques in software measurement tools and development environments. Developers may benefit from this and provide feedback whereas they have available conceptual cohesion metrics within their daily working tool environment.

BIBLIOGRAPHY

- Adams, B.; Jiang, Z. M.; Hassan, A. E. Identifying crosscutting concerns using historical code changes. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE'10), p. 305–314. ISBN 978-1-60558-719-6.
- Antoniol, G.; Gueheneuc, Y. G. Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 32, p. 627–641, 2006.
- Badri, L.; Badri, M.; Gueye, A. B. Revisiting class cohesion: An empirical investigation on several systems. *Journal of Object Technology*, v. 7, n. 6, p. 55–75, 2008.
- Baldi, P. F. et al. A theory of aspects as latent topics. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. New York, NY, USA: ACM, 2008. (OOPSLA '08), p. 543–562.
- Basili, V.; Briand, L.; Melo, W. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, v. 22, n. 10, p. 751–761, Oct 1996.
- Batista, E.; Sant'Anna, C. Avaliação experimental da relação de coesão e acoplamento com o esforço de compreensão de software. In: *2nd Latin-American School on Software Engineering*. Porto Alegre, Brazil: [s.n.], 2015.
- Bavota, G. et al. An empirical study on the developers' perception of software coupling. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 692–701.
- Bieman, J. M.; Kang, B.-K. Cohesion and reuse in an object-oriented system. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 20, n. SI, p. 259–262, ago. 1995. ISSN 0163-5948.
- Biggerstaff, T. J.; Mitbander, B. G.; Webster, D. E. Program understanding and the concept assignment problem. *Commun. ACM*, ACM, New York, NY, USA, v. 37, n. 5, p. 72–82, maio 1994.
- Binkley, D. et al. Understanding lda in source code analysis. In: *Proceedings of the 22Nd International Conference on Program Comprehension*. New York, NY, USA: ACM, 2014. (ICPC 2014), p. 26–36.

Blei, D. M.; Ng, A. Y.; Jordan, M. I. Latent dirichlet allocation. *J. Mach. Learn. Res.*, JMLR.org, v. 3, p. 993–1022, mar. 2003. ISSN 1532-4435.

Bohnet, J.; Voigt, S.; Dollner, J. Locating and understanding features of complex software systems by synchronizing time-, collaboration- and code-focused views on execution traces. In: *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. [S.l.: s.n.], 2008. p. 268 –271.

Breiman, L. et al. *Classification and Regression Trees*. New York: Chapman and Hall, 1984.

Breu, S.; Zimmermann, T. Mining aspects from version history. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 221–230, 2006. ISSN 1527-1366.

Briand, L.; Bunse, C.; Daly, J. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *Software Engineering, IEEE Transactions on*, v. 27, n. 6, p. 513 –530, jun 2001.

Briand, L. C.; Daly, J. W.; Wüst, J. A unified framework for cohesion measurement in object-oriented systems. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 3, n. 1, p. 65–117, jul. 1998. ISSN 1382-3256.

Briand, L. C.; Morasca, S.; Basili, V. R. Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 25, p. 722–743, 1999. ISSN 0098-5589.

Briand, L. C. et al. Exploring the relationship between design measures and software quality in object-oriented systems. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 51, n. 3, p. 245–273, maio 2000.

Buse, R. P.; Weimer, W. R. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 36, n. 4, p. 546–558, 2010.

Chae, H. S.; Kwon, Y. R.; Bae, D.-H. A cohesion measure for object-oriented classes. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 30, n. 12, p. 1405–1431, out. 2000. ISSN 0038-0644.

Chen, Z. et al. A novel approach to measuring class cohesion based on dependence analysis. In: *Software Maintenance, 2002. Proceedings. International Conference on*. [S.l.: s.n.], 2002. p. 377 – 384.

Chidamber, S. R.; Kemerer, C. F. Towards a metrics suite for object oriented design. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 26, n. 11, p. 197–211, nov. 1991.

Chidamber, S. R.; Kemerer, C. F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 20, n. 6, p. 476–493, jun. 1994. ISSN 0098-5589.

Chowdhury, I.; Zulkernine, M. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2010. (SAC '10), p. 1963–1969.

Cohen, J. *Statistical Power Analysis for the Behavioral Sciences (2nd Edition)*. 2. ed. [S.l.]: Routledge Academic, 1988. Hardcover.

Counsell, S. et al. Object-oriented cohesion subjectivity amongst experienced and novice developers: an empirical study. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 31, n. 5, p. 1–10, set. 2006.

Cox, G. W.; Etkorn, L. H.; Jr., W. E. H. Cohesion metric for object-oriented systems based on semantic closeness from disambiguity. *Applied Artificial Intelligence*, Taylor & Francis, p. 419–436, 2006.

Dallal, J. A.; Briand, L. C. A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 21, n. 2, p. 8:1–8:34, mar. 2012.

Deerwester, S. et al. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, v. 41, n. 6, p. 391–407, 1990.

Dit, B. et al. Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice*, John Wiley and Sons, 2011.

Dit, B.; Revelle, M.; Poshyvanyk, D. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 18, n. 2, p. 277–309, abr. 2013.

Eaddy, M. *An empirical assessment of the crosscutting concern problem*. Tese (Doutorado), New York, NY, USA, 2008. AAI3305217.

Eaddy, M. et al. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. [S.l.: s.n.], 2008. p. 53–62.

Eaddy, M. et al. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 34, p. 497–515, 2008.

Eisenberg, A.; Volder, K. D. Dynamic feature traces: finding features in unfamiliar code. In: *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. [S.l.: s.n.], 2005. p. 337–346.

Etkorn, L.; Delugach, H. Towards a semantic metrics suite for object-oriented design. In: *Proceedings of the Technology of Object-Oriented Languages and Systems*. Washington, DC, USA: IEEE Computer Society, 2000. (TOOLS '00), p. 71–. ISBN 0-7695-0774-3. Disponível em: <http://dl.acm.org/citation.cfm?id=832261.833277>).

Etzkorn, L. H. et al. A comparison of cohesion metrics for object-oriented systems. *Information and Software Technology*, v. 46, n. 10, p. 677 – 687, 2004.

Falleri, J.-R. et al. Efficient retrieval and ranking of undesired package cycles in large software systems. In: Bishop, J.; Vallecillo, A. (Ed.). *Objects, Models, Components, Patterns*. [S.l.]: Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6705). p. 260–275.

Feigenspan, J. et al. Measuring programming experience. In: *Proceedings of the 20th International Conference on Program Comprehension (ICPC)*. Los Alamitos, CA: [s.n.], 2012. p. 73–82.

Fenton, N. E.; Pfleeger, S. L. *Software Metrics: A Rigorous and Practical Approach*. 2nd. ed. Boston, MA, USA: PWS Publishing Co., 1998. ISBN 0534954251.

Figueiredo, E. *Concern-Oriented Heuristic Assessment of Design Stability*. Tese (Doutorado), Lancaster, UK, 2009.

Figueiredo, E. et al. Evolving software product lines with aspects: an empirical study on design stability. In: *Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008. (ICSE '08), p. 261–270.

Figueiredo, E. et al. On the impact of crosscutting concern projection on code measurement. In: *Proceedings of the tenth international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2011. (AOSD '11), p. 81–92.

Figueiredo, E. et al. Applying and evaluating concern-sensitive design heuristics. *Journal of Systems and Software*, v. 85, n. 2, p. 227 – 243, 2012.

Fleiss, J. L. Measuring Nominal Scale Agreement Among Many Raters. *Psychological Bulletin*, v. 76, n. 5, p. 378–382, 1971.

Garcia, A. et al. Modularizing design patterns with aspects: a quantitative study. In: *Proceedings of the 4th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2005. (AOSD '05), p. 3–14.

Grant, S.; Cordy, J. R. Estimating the optimal number of latent concepts in source code analysis. *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 65–74, 2010.

Grant, S.; Cordy, J. R.; Skillicorn, D. B. Using topic models to support software maintenance. In: *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2012. (CSMR '12), p. 403–408.

Gwet, K. *Handbook of Inter-Rater Reliability (3rd Edition): The Definitive Guide to Measuring the Extent of Agreement Among Multiple Raters*. 3. ed. [S.l.]: Advanced Analytics Press, 2012.

- Hashimoto, M.; Mori, A. Enhancing history-based concern mining with fine-grained change analysis. In: *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. [S.l.: s.n.], 2012. p. 75–84.
- Henderson-sellers, B. *Software Metrics*. [S.l.]: Prentice Hall, UK, 1996.
- Hill, E.; Rao, S.; Kak, A. On the use of stemming for concern location and bug localization in java. In: *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*. [S.l.: s.n.], 2012. p. 184–193.
- Hitz, M.; Montazeri, B. Measuring coupling and cohesion in object-oriented systems. In: *Proc. Intl. Sym. on Applied Corporate Computing*. [S.l.: s.n.], 1995.
- Jolliffe, I. T. *Principal Component Analysis*. Second. [S.l.]: Springer, 2002. Hardcover. ISBN 0387954422.
- Kabaili, H.; Keller, R. K.; Lustman, F. Cohesion as changeability indicator in object-oriented systems. *15th European Conference on Software Maintenance and Reengineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 39, 2001.
- Katzmarski, B.; Koschke, R. Program complexity metrics and programmer opinions. In: *Proc. 20th IEEE International Conference on Program Comprehension*. [S.l.]: IEEE, 2012. p. 17–26.
- Kellens, A.; Mens, K.; Tonella, P. Transactions on aspect-oriented software development iv. In: Rashid, A.; Aksit, M. (Ed.). Berlin, Heidelberg: Springer-Verlag, 2007. cap. A survey of automated code-level aspect mining techniques, p. 143–162. ISBN 3-540-77041-0, 978-3-540-77041-1. Disponível em: (<http://dl.acm.org/citation.cfm?id=1793854.1793862>).
- Koru, A. G.; Liu, H. Identifying and characterizing change-prone classes in two large-scale open-source products. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 80, n. 1, p. 63–73, jan. 2007.
- Koru, A. G.; Tian, J. J. Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 31, p. 625–642, 2005.
- Landis, J. R.; Koch, G. G. The Measurement of Observer Agreement for Categorical Data. *Biometrics*, v. 33, n. 1, p. 159–174, mar. 1977.
- Lee, Y. S.; Liang, B. S. Measuring the coupling and cohesion of an object-oriented program based on information flow. In: *Prof. Intl. Conference on Software Quality*. [S.l.: s.n.], 1995.
- Linstead, E. et al. Mining concepts from code with probabilistic topic models. In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2007. (ASE '07), p. 461–464.

Liu, Y. et al. Modeling class cohesion as mixtures of latent topics. In: *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*. [S.l.]: IEEE, 2009. p. 233–242.

Lu, H. et al. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical Softw. Engg.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 17, n. 3, p. 200–242, jun. 2012.

Lucia, A. D. et al. Using ir methods for labeling source code artifacts: Is it worthwhile? In: *ICPC*. [S.l.: s.n.], 2012. p. 193–202.

Madhavji, N. H.; Fernandez-ramil, J.; Perry, D. *Software Evolution and Feedback: Theory and Practice*. [S.l.]: John Wiley & Sons, 2006. ISBN 0470871806.

Mäntylä, M. V.; Lassenius, C. Drivers for software refactoring decisions. In: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. New York, NY, USA: ACM, 2006. (ISESE '06), p. 297–306.

Marcus, A.; Poshyvanyk, D. The conceptual cohesion of classes. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2005. (ICSM '05), p. 133–142.

Marcus, A.; Poshyvanyk, D.; Ferenc, R. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 34, p. 287–300, 2008.

Marcus, A. et al. An information retrieval approach to concept location in source code. In: *Proceedings of the 11th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2004. (WCRE '04), p. 214–223. ISBN 0-7695-2243-2. Disponível em: <http://dl.acm.org/citation.cfm?id=1038267.1039053>.

Martin, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN 0135974445.

Maskeri, G.; Sarkar, S.; Heafield, K. Mining business topics in source code using latent dirichlet allocation. In: *Proceedings of the 1st India Software Engineering Conference*. New York, NY, USA: ACM, 2008. (ISEC '08), p. 113–120.

Myers, G. *Composite/Structured Design*. 1st. ed. New York, NY, USA: Van Nostrand Reinhold, 1978. ISBN 0442805845.

Neave, H. *Elementary Statistics Tables*. 2nd. ed. [S.l.]: Routledge, 2010.

Nguyen, T. T. et al. Aspect recommendation for evolving software. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011. (ICSE '11), p. 361–370.

- Poshyvanyk, D.; Marcus, A. Combining formal concept analysis with information retrieval for concept location in source code. In: *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*. [S.l.: s.n.], 2007. p. 37–48.
- Punter, T. et al. Conducting on-line surveys in software engineering. In: *Proceedings of the 2003 International Symposium on Empirical Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003. (ISESE '03), p. 80–.
- Robillard, M. P.; Murphy, G. C. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 16, n. 1, fev. 2007.
- Romano, D.; Pinzger, M. Using source code metrics to predict change-prone java interfaces. *Software Maintenance, IEEE International Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 303–312, 2011.
- Rysselberghe, F. V.; Demeyer, S. Studying versioning information to understand inheritance hierarchy changes. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007. (MSR '07), p. 16–.
- Sant'Anna, C. *On the Modularity of Aspect-Oriented Design: A Concern-Driven Measurement Approach*. Tese (Doutorado), Rio de Janeiro, Brazil, 2008.
- Sant'Anna, C. et al. On the modularity of software architectures: A concern-driven measurement framework. In: Oquendo, F. (Ed.). *Software Architecture*. [S.l.]: Springer Berlin / Heidelberg, 2007, (Lecture Notes in Computer Science, v. 4758). p. 207–224. ISBN 978-3-540-75131-1.
- Sartipi, K.; Safyallah, H. Dynamic knowledge extraction from software systems using sequential pattern mining. *International Journal of Software Engineering and Knowledge Engineering*, p. 761–782, 2010.
- Savage, T. et al. Topicxp: Exploring topics in source code using latent dirichlet allocation. In: *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2010. (ICSM '10), p. 1–6.
- Seaman, C. B. Qualitative methods in empirical studies of software engineering. *IEEE Trans Softw. Eng.*, IEEE Computer Society, Los Alamitos, CA, USA, v. 25, p. 557–572, 1999.
- Shull, F.; Singer, J.; Sjberg, D. I. K. *Guide to Advanced Empirical Software Engineering*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010.
- Silva, B. *Developers reasoning about module cohesion: companion website*. out. 2013. Disponível em: <http://homes.dcc.ufba.br/~brunocs/CohesionSurveyResults.html>.
- Silva, B. *Conceptual cohesion and change-proneness: study resources*. jun. 2015. Disponível em: http://homes.dcc.ufba.br/~brunocs/ICPC12_extension.html.

Silva, B. *The Impact of Different Concern Mapping Strategies on Conceptual Cohesion: study resources*. jun. 2015. Disponível em: <http://homes.dcc.ufba.br/~brunocs/IST2015-DiffMappings.html>.

Silva, B. et al. Concern-based cohesion: Unveiling a hidden dimension of cohesion measurement. In: *Proc. 20th IEEE International Conference on Program Comprehension*. [S.l.]: IEEE, 2012. p. 103–112.

Silva, L. et al. Does javascript software embrace classes? In: *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. [S.l.: s.n.], 2015. p. 73–82.

Smith, E. et al. Improving developer participation rates in surveys. In: *Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering*. [S.l.: s.n.], 2013.

Stevens, W. P.; Myers, G. J.; Constantine, L. L. Structured design. *IBM Systems Journal*, v. 13, n. 2, p. 115–139, 1974.

Tarr, P. et al. N degrees of separation: multi-dimensional separation of concerns. In: *ICSE '99: 21st International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999. p. 107–119. ISBN 1-58113-074-0.

Terceiro, A. et al. Análizo: an extensible multi-language source code analysis and visualization toolkit. In: *1st Brazilian Conference on Software - Tools track*. Washington, DC, USA: IEEE Computer Society, 2010.

Terra, R. et al. Qualitas.class corpus: A compiled version of the qualitas corpus. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 38, n. 5, p. 1–4, ago. 2013.

Thomas, S. W. et al. Validating the use of topic models for software evolution. In: *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. Washington, DC, USA: IEEE Computer Society, 2010. (SCAM '10), p. 55–64.

Together, B. *Borland Together IDE*. jun. 2011. Disponível em: <http://www.borland.com/us/products/together>.

Trifu, M. Improving the dataflow-based concern identification approach. *2011 15th European Conference on Software Maintenance and Reengineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 109–118, 2009.

Ujhazi, B. et al. New conceptual coupling and cohesion metrics for object-oriented systems. In: *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. Washington, DC, USA: IEEE Computer Society, 2010. (SCAM '10), p. 33–42.

Yourdon, E.; Constantine, L. L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. 1st. ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1979. ISBN 0138544719.

Appendices



R SCRIPT FOR PCA

Listing A.1 R script for executing PCA.

```
library(psych) # library for the 'principal()' function

#the distributions analyzed here do not include CC (change count) metric.
  Only cohesion metrics

TOMCAT <-
  read.table("cohesionMetricsFinal_with_MWE_tomcat_without_LCbC_Topics.csv",
    header=TRUE, sep=",");
TOMCAT2 <- TOMCAT[,c(2,3,4,5,6,7,8)] # keeping only desired columns
TOMCAT2 <- na.omit(TOMCAT2) # omitting N/A measures

TOMCAT_PCA2 <- principal(TOMCAT2, nfactors=5, rotate="varimax") # PCA
  function with varimax rotation
TOMCAT_PCA2

JFREECHART <-
  read.table("cohesionMetricsFinal_with_MWE_tomcat_without_LCbC_Topics.csv",
    header=TRUE, sep=",");
JFREECHART2 <- JFREECHART[,c(2,3,4,5,6,7,8)]
JFREECHART2 <- na.omit(JFREECHART2)

JFREECHART_PCA2 <- principal(JFREECHART2, nfactors=5, rotate="varimax")
JFREECHART_PCA2

FINDBUGS <-
  read.table("cohesionMetricsFinal_with_MWE_Findbugs_without_LCbC_Topics.csv",
    header=TRUE, sep=",");
FINDBUGS2 <- FINDBUGS[,c(2,3,4,5,6,7,8)]
FINDBUGS2 <- na.omit(FINDBUGS2)

FINDBUGS_PCA2 <- principal(FINDBUGS2, nfactors=5, rotate="varimax")
FINDBUGS_PCA2

FREECOL <-
  read.table("cohesionMetricsFinal_with_MWE_Freecol_without_LCbC_Topics.csv",
    header=TRUE, sep=",");
FREECOL2 <- FREECOL[,c(2,3,4,5,6,7,8)]
FREECOL2 <- na.omit(FREECOL2)

FREECOL_PCA2 <- principal(FREECOL2, nfactors=5, rotate="varimax")
FREECOL_PCA2
```

```
JEDIT <-  
  read.table("cohesionMetrics_MERGED_V3_jEdit-4.3.2_without_LCbC_MM_Topics.csv",  
    header=TRUE, sep=",");  
JEDIT2 <- JEDIT[,c(2,3,4,5,6,7,8)]  
JEDIT2 <- na.omit(JEDIT2)  
  
JEDIT_PCA2 <- principal(JEDIT2, nfactors=5, rotate="varimax")  
JEDIT_PCA2  
  
RHINO <-  
  read.table("cohesionMetrics_MERGED_V3_Rhino_1.6R5_without_LCbC_MM_Topics.csv",  
    header=TRUE, sep=",");  
RHINO2 <- RHINO[,c(2,3,4,5,6,7,8)]  
RHINO2 <- na.omit(RHINO2)  
  
RHINO_PCA2 <- principal(RHINO2, nfactors=5, rotate="varimax")  
RHINO_PCA2
```

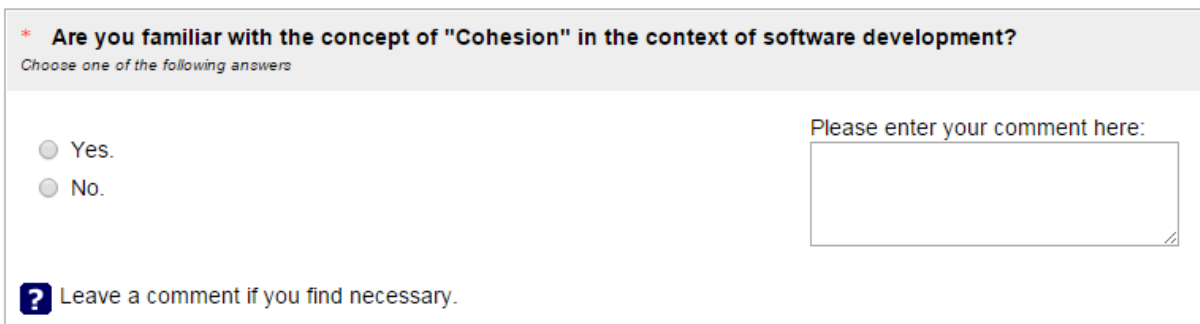



SURVEY QUESTIONNAIRE

The following figures reproduce the questions we had in our web-based survey about developers' opinion on module cohesion. The figures show the exact questions participants had to answer according to the following sequence. The first question (Figure B.1) is mandatory and starts the survey. Then the second question (Figure B.2) appeared only in case of "yes" answer in first question. Afterwards, we showed a page defining cohesion (Figure B.3).

The next questions are illustrated in Figures B.4 to B.6. They show questionnaire pages for the three scenarios of cohesion comparison and rating, including a mandatory explanation about the applied rationale. The last group of questions (Figure B.7) is intended to collect participant's profile, with all of them mandatory except the last question.

It is worth noting that this is a web-based survey. Therefore, we had page interactions such as automatic verification of mandatory questions, "save & resume later" feature, progress bar, and hyperlinks to the classes source code.



* **Are you familiar with the concept of "Cohesion" in the context of software development?**
Choose one of the following answers

Yes.
 No.

Please enter your comment here:


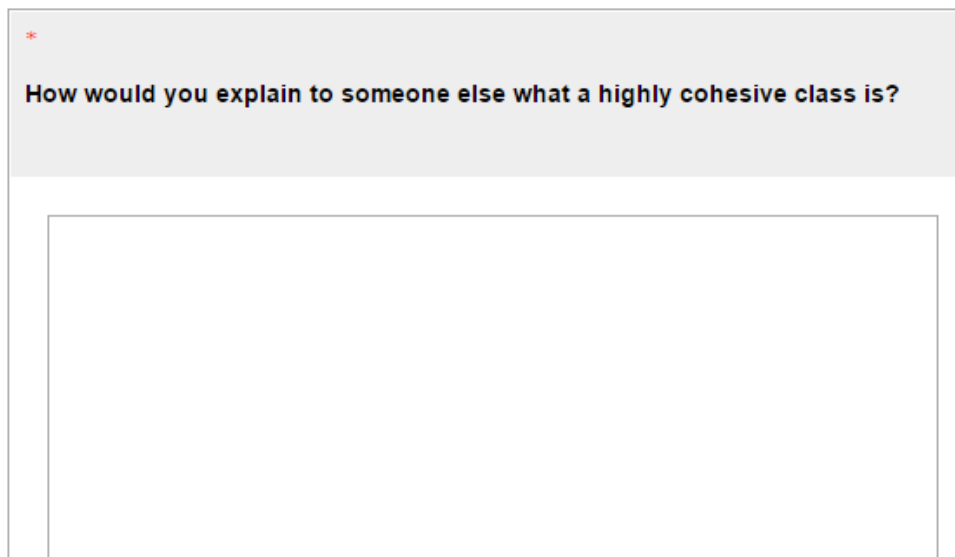
 Leave a comment if you find necessary.

Figure B.1 Cohesion familiarity



*
How would you explain to someone else what a highly cohesive class is?

Figure B.2 Cohesion explanation

Cohesion Definition (This is just an explanation. Please, read carefully. There is nothing to answer)

The term **cohesion** was first introduced in 1974 by Stevens and colleagues in the context of structured design and defined as **a measure of the degree to which the internal elements* of a module* belong together**. Accordingly, **in a highly cohesive module, all the elements are related to the performance of a single behavior of the software**.

Martin (2003) defined that **a highly cohesive class should have a single responsibility**. Bieman and Kang (1995) stated that **a highly cohesive class should be difficult to split**.

**A module can be considered as a class in object-oriented programming. Accordingly, the internal elements are its fields and methods.*

?

Stevens, W. P.; Myers, G. J.; Constantine, L. L. Structured design. IBM Systems Journal, v. 13, n. 2, p. 115-139, 1974.

Martin, R. C. Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

Bieman, J. M.; Kang, B.-K. Cohesion and reuse in an object-oriented system. SIGSOFT Softw. Eng. Notes, ACM, New York, NY, USA, v. 20, n. SI, p. 259-262, 1995.

Figure B.3 Information about cohesion definition

Consider the following two Java classes that belongs to a hotel management system.

The system supports the management of a hotel, i.e. creating/canceling/editing of bookings, billing, creating/editing rooms/categories/users and storing guest data.

These two classes are utility classes for the data base access.

The first one, [DB_Backend.java \(click here to access the source code\)](#), is responsible for managing data base connections.
[\(Click here if you want to download the java source code to open on your favorite editor\)](#)

The second one, [DB_InsertUpdate.java \(click here to access the source code\)](#), is responsible for writing data to the data base either by inserting or by updating information.
[\(Click here if you want to download the java source code to open on your favorite editor\)](#)

The Java classes were taken from a real and open source software hosted at SourceForge (<http://fgmp-hm.sourceforge.net/>).

Please, analyze them and regarding their cohesion
Choose one of the following answers

- The first class ([DB_Backend](#)) is more cohesive than the second one ([DB_InsertUpdate](#))
- The second class ([DB_InsertUpdate](#)) is more cohesive than the first one ([DB_Backend](#))
- Both classes have quite similar cohesion.
- I don't know.

? Take all the time you need to understand the classes and give your answer. Response time is not under consideration.

*** Why did you pick that option? What was your rationale to answer this question?**

Figure B.4 First scenario about cohesion comparison and rating

Consider the following two Java classes that belongs to the *FamilyTree* project.

This Java project implements family relationships between husband and wife, and between parents and their children.

The relationships may be stored in disk and also they may be displayed in a graphical user interface.

[RelationSpouse.java \(click here to access the source code\)](#): represents the relationship between husband and wife.
([Click here if you want to download the java source code to open on your favorite editor](#))

[RelationParentChild.java \(click here to access the source code\)](#): represents the relationship between a parent and his child.
([Click here if you want to download the java source code to open on your favorite editor](#))

The Java classes were taken from an open source software hosted at <http://www.soberit.tkk.fi/~mmantyla/SESE2006/>.

Please, analyze them and regarding their cohesion
Choose one of the following answers

- The first class (*RelationSpouse*) is more cohesive than the second one (*RelationParentChild*)
- The second class (*RelationParentChild*) is more cohesive than the first one (*RelationSpouse*)
- Both classes have quite similar cohesion.
- I don't know.

? Take all the time you need to understand the classes and give your answer. Response time is not under consideration.

*** Why did you pick that option? What was your rationale to answer this question?**

Figure B.5 Second scenario about cohesion comparison and rating

Consider the following two Java classes that belongs to the *FamilyTree* project.

This Java project implements family relationships between husband and wife, and between parents and their children.

The relationships may be stored in disk and also they may be displayed in a graphical user interface.

[RelationSpouse.java \(click here to access the source code\)](#): represents the relationship between husband and wife.
(Click here if you want to download the java source code to open on your favorite editor)

[RelationParentChild.java \(click here to access the source code\)](#): represents the relationship between a parent and his child.
(Click here if you want to download the java source code to open on your favorite editor)

The Java classes were taken from an open source software hosted at <http://www.soberit.tkk.fi/~mmantyla/ISESE2006/>.

Please, analyze them and regarding their cohesion
Choose one of the following answers

- The first class (***RelationSpouse***) is more cohesive than the second one (***RelationParentChild***)
- The second class (***RelationParentChild***) is more cohesive than the first one (***RelationSpouse***)
- Both classes have quite similar cohesion.
- I don't know.

? Take all the time you need to understand the classes and give your answer. Response time is not under consideration.

* Why did you pick that option? What was your rationale to answer this question?

Figure B.6 Third scenario about cohesion comparison and rating

*** What is your academic degree?**
Choose one of the following answers

- Undergraduate
- Graduate
- Graduate (Certified expert)
- Graduate (Master)
- Graduate (PhD)
- No academic degree

*** What is your job position?**
Check any that apply

- Software developer
- System analyst
- Tester
- Software architect/designer
- Business analyst
- Project manager
- Professor/Lecturer
- Researcher
- Student
- Unemployed or Retired
- Other:

*** On a scale from 1 (Very Unexperienced) to 10 (Very Experienced), how do you measure your programming experience?**

	1	2	3	4	5	6	7	8	9	10
My programming experience.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

*** Related to your programming experience, please answer from 1 (Very Unexperienced) to 5 (Very Experienced):**

	1	2	3	4	5
How do you estimate your programming experience compared to your colleagues?	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
How experienced are you with the Java programming language?	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
How experienced are you with the object-oriented programming paradigm?	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

*** For how many years have you been programming?**
Only numbers may be entered in this field.

*** For how many years have you been programming for large software projects, e.g. in a company?**
Only numbers may be entered in this field.

Do you have any comments regarding this study? Feel free to ask, suggest or criticize anything you want. (Optional)

Figure B.7 Participant profile



SURVEY CLASSES

Listing C.1 DB_Backend class source code.

```
/*
 * This file is part of FGMP-Hotelverwaltung
 *
 * Copyright 2010, 2009 Daniel Fischer, David Gawehn, Martin Meyer, Christian
 * Pusch
 *
 * FGMP-Hotelverwaltung is free software: you can redistribute it and/or
 * modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package FGMP_Hotel_Management.Datenbank2;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;
import FGMP_Hotel_Management.Messaging;

/**
 * Provide methods for the connection to the database
 *
 * @author Daniel Fischer, David Gawehn
 */
public class DB_Backend {

    private final static int conns= 5;
    public static Connection [] con_pool = new Connection [conns];

    private static int concount=-1;
    private String Host;
    private String Port;
```



```
private String Database;
private String User;
private String Password;

public DB_Backend(String h, String p, String d, String u, String pw) {
    init(h, p, d, u, pw);
}

public void init(String h, String p, String d, String u, String pw) {
    Host = h;
    Port = p;
    Database = d;
    User = u;
    Password = pw;
}

/**
 * Provide the central connection to the MySQL database
 *
 * @param host      Database computer host
 * @param database  DB name
 * @param user      DB-User
 * @param PW        User-Password
 * @return         false, if failed
 */
public static Boolean connect_DB(String host, String database, String
    user, String PW) {

    Boolean connected = false;

    try {
        Class.forName("com.mysql.jdbc.Driver");

        try {
            for (int i=0; i<5; i++) {
                con_pool[i] =
                    DriverManager.getConnection("jdbc:mysql://" + host +
                        "/" + database, user, PW);
            }
            connected = true;
        } catch (SQLException ex) {
            Messaging.show_Dialog(ex.toString().substring(ex.toString().
                indexOf(".")+1), "Error", JOptionPane.ERROR_MESSAGE);
        }
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

```
    }
    return connected;
}

/**
 * Provide the central connection to the MySQL database
 */
public Boolean connect_DB() {

    Boolean connected = false;

    try {
        Class.forName("com.mysql.jdbc.Driver");

        try {
            for (int i=0; i<conns; i++) {
                con_pool[i] =
                    DriverManager.getConnection("jdbc:mysql://" +
                        Host.concat(":").concat(Port).concat("/").
                        concat(Database), User, Password);
            }
            connected = true;

        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return connected;
}

/**
 * Closing the database connection
 */
public static void close_DB() {
    try {
        for (int i=0; i<conns; i++) {
            if (con_pool[i]!=null)
                con_pool[i].close();
        }
    } catch (SQLException ex) {
        Logger.getLogger(DB_Backend.class.getName()).log(Level.SEVERE,
            null, ex);
    }
}
```

```
/**
 * Obtain a connection from the connection pool
 */
public static Connection getConnection() {

    if (concount < conns-1) {
        concount++;
    } else {
        concount=0;
    }
    return con_pool[concount];
}
}
```

Listing C.2 DB_InsertUpdate class source code.

```
/*
 * This file is part of FGMP-Hotelverwaltung
 *
 * Copyright 2010, 2009 Daniel Fischer, David Gawehn, Martin Meyer, Christian
 * Pusch
 *
 * FGMP-Hotelverwaltung is free software: you can redistribute it and/or
 * modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package FGMP_Hotel_Management.Datenbank2;

import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```

/**
 * Provide methods to write (insert/update) to the database
 *
 * @author Daniel Fischer, David Gawehn
 */
public class DB_InsertUpdate {

    /**
     * Create a new record in a table
     *
     * @param Table    Target table to be written
     * @param Values  ArrayList of data, length of the list must be the same
     *                number of table columns
     * @return        1, when written, -1 if not written
     */
    public static int insertAt (String Table, ArrayList Values) {

        try {
            String values = "";
            for (int i = 0; i < Values.size() - 1; i++) {
                values = values.concat("'" + String.valueOf(Values.get(i)) +
                    "',");
            }
            values = values.concat("'" +
                String.valueOf(Values.get(Values.size() - 1)) + "'");
            Statement statement = DB_Backend.getConnection().createStatement();
            int insertAt = statement.executeUpdate("INSERT INTO " + Table + "
                VALUES (" + values + ")");
            return insertAt;

        } catch (SQLException ex) {
            Logger.getLogger(DB_InsertUpdate.class.getName()).log(Level.SEVERE,
                null, ex);
            return -1;
        }
    }

    /**
     * Update the record in a table where the column "where" and the value of
     * "what" are given
     *
     * @param Table    target table
     * @param Values  the values in the form: "Column Name", value, "Column
     *                Name", value...
     * @param where    column for the where clause
     */

```

```

* @param what    value for the condition
* @return       1, when successful; -1, when not
*/
public static int update (String Table, ArrayList Values, String where,
    int what) {
    try {
        String values = "";
        for (int i = 0; i < Values.size() - 2; i += 2) {
            values = values.concat(String.valueOf(Values.get(i)) + "=" +
                String.valueOf(Values.get(i + 1)) + ", ");
        }
        values = values.concat(String.valueOf(Values.get(Values.size() -
            2)) + "=" + String.valueOf(Values.get(Values.size() - 1)) +
            "");
        PreparedStatement statement =
            DB_Backend.getConnection().prepareStatement("UPDATE " + Table +
                " SET " + values + " WHERE " + where + "= ? LIMIT 1");
        statement.setInt(1,what);
        int update = statement.executeUpdate();
        return update;
    } catch (SQLException ex) {
        Logger.getLogger(DB_InsertUpdate.class.getName()).log(Level.SEVERE,
            null, ex);
        return -1;
    }
}

public static int update (String Table, ArrayList Values, String where,
    String what) {
    try {
        String values = "";
        for (int i = 0; i < Values.size() - 2; i += 2) {
            values = values.concat(String.valueOf(Values.get(i)) + "=" +
                String.valueOf(Values.get(i + 1)) + ", ");
        }
        values = values.concat(String.valueOf(Values.get(Values.size() -
            2)) + "=" + String.valueOf(Values.get(Values.size() - 1)) +
            "");
        PreparedStatement statement =
            DB_Backend.getConnection().prepareStatement("UPDATE " + Table +
                " SET " + values + " WHERE " + where + "= ? LIMIT 1");
        statement.setString(1,what);
        int update = statement.executeUpdate();
        return update;
    } catch (SQLException ex) {
        Logger.getLogger(DB_InsertUpdate.class.getName()).log(Level.SEVERE,
            null, ex);
    }
}

```

```
        return -1;
    }
}
}
```

Listing C.3 Main_Config2 class source code.

```
/*
 * This file is part of FGMP-Hotelverwaltung
 *
 * Copyright 2011, 2010, 2009 Daniel Fischer, David Gawehn, Martin Meyer,
 * Christian Pusch
 *
 * FGMP-Hotelverwaltung is free software: you can redistribute it and/or
 * modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package FGMP_Hotel_Management.Configuration;

import FGMP_Hotel_Management.Language.ErrorMessage;
import FGMP_Hotel_Management.Language.LanguageFile;
import FGMP_Hotel_Management.Messaging;
import java.beans.XMLDecoder;
import java.beans.XMLEncoder;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;
```

```
/**
 * Get and set information in the main config file
 *
 * @author David Gawehn
 */
public class Main_Config2 {
    private String fileName;
    private XMLDecoder d;
    private XMLEncoder o;

    private String hotelAddress;
    private String fax;
    private String telephone;
    private String mail;
    private String web;
    private String VAT;
    private String invoice_Place;
    private String invoice_extD;
    private Vector[] cancellationFee = new Vector[2];

    private String DB_Host;
    private String DB_Port;
    private String DB_Name;
    private String DB_User;
    private String DB_Password;

    private String Currency;
    private String Bill_Top;
    private String Bill_Bottom;

    private String languageFileAbsolutePath;

    public Main_Config2(String fn) throws Exception{
        this.fileName = fn;
        this.init(fn);
    }

    private void init(String fn) throws Exception{
        this.load_config();
    }

    private void load_config() throws FileNotFoundException,
        ArrayIndexOutOfBoundsException{
        d = new XMLDecoder(new BufferedInputStream(new
            FileInputStream(this.fileName)));

        this.hotelAddress = (String) d.readObject();
    }
}
```

```
this.fax = (String) d.readObject();
this.telephone = (String) d.readObject();
this.mail = (String) d.readObject();
this.web = (String) d.readObject();
this.VAT = (String) d.readObject();
this.invoice_Place = (String) d.readObject();
this.invoice_extD = (String) d.readObject();
this.cancellationFee[0] = (Vector)d.readObject();
this.cancellationFee[1] = (Vector)d.readObject();
this.DB_Name = (String) d.readObject();
this.DB_Host = (String) d.readObject();
this.DB_User = (String) d.readObject();
this.DB_Port = (String) d.readObject();
this.DB_Password = (String) d.readObject();
this.Currency = (String) d.readObject();
this.Bill_Top = (String) d.readObject();
this.Bill_Bottom = (String) d.readObject();

String pa = (String) d.readObject();
if (pa != null) {
    LanguageFile.applyLanguageFile(new File(pa));
}
}

public void save_config() throws FileNotFoundException{
    this.o = new XMLEncoder(new BufferedOutputStream(new
        FileOutputStream(this.fileName)));
    o.writeObject(hotelAddress);
    o.writeObject(fax);
    o.writeObject(telephone);
    o.writeObject(mail);
    o.writeObject(web);
    o.writeObject(VAT);
    o.writeObject(invoice_Place);
    o.writeObject(invoice_extD);
    o.writeObject(cancellationFee[0]);
    o.writeObject(cancellationFee[1]);
    o.writeObject(DB_Name);
    o.writeObject(DB_Host);
    o.writeObject(DB_User);
    o.writeObject(DB_Port);
    o.writeObject(DB_Password);
    o.writeObject(Currency);
    o.writeObject(Bill_Top);
    o.writeObject(Bill_Bottom);
    o.writeObject(languageFileAbsolutePath);
    o.flush();
}
```



```
    }

    /*      *
    * SETs *
    *      */
    public void setBill_Bottom(String Bill_Bottom) {
        this.Bill_Bottom = Bill_Bottom;
    }

    public void setBill_Top(String Bill_Top) {
        this.Bill_Top = Bill_Top;
    }

    public void setCurrency(String Currency) {
        this.Currency = Currency;
    }

    public void setLanguageFileAbsolutePath(String LanguageFileAbsolutePath) {
        this.languageFileAbsolutePath = LanguageFileAbsolutePath;
    }

    public void setVAT(String vat) {
        this.VAT = vat;
    }

    public void setInvoice_Place(String invoice_Place) {
        this.invoice_Place = invoice_Place;
    }

    public void setInvoice_extD(String invoice_extD) {
        this.invoice_extD = invoice_extD;
    }

    public void setCancellationFee(Vector[] cancellation) {
        this.cancellationFee = cancellation;
    }

    public void setHost(String Host) {
        this.DB_Host = Host;
    }

    public void setPassword(String PSWD) {
        this.DB_Password = PSWD;
    }

    public void setPort(String Port) {
        this.DB_Port = Port;
    }
}
```

```
}

public void setUser(String User) {
    this.DB_User = User;
}

public void setDatabase(String DBName) {
    this.DB_Name = DBName;
}

public void setHotelAddress(String text) {
    this.hotelAddress = text;
}

public void setTelephone(String text) {
    this.telephone = text;
}

public void setFax(String text) {
    this.fax = text;
}

public void setMail(String text) {
    this.mail = text;
}

public void setWeb(String text) {
    this.web = text;
}

/*      *
 * GETs *
 *      */
public String getLanguageFileAbsolutePath() {
    return languageFileAbsolutePath;
}

public Vector[] getCancellationFee() {
    return cancellationFee;
}

public String getDB_Host() {
    return this.DB_Host;
}

public String getDB_Port() {
    return this.DB_Port;
}
}
```

```
public String getDB_Name() {
    return this.DB_Name;
}

public String getDB_Password() {
    return this.DB_Password;
}

public String getDB_User() {
    return this.DB_User;
}

public String getHotelAddress() {
    return this.hotelAddress;
}

public String getFax() {
    return this.fax;
}

public String getTelephone() {
    return this.telephone;
}

public String getMail() {
    return this.mail;
}

public String getWeb() {
    return this.web;
}

public String getVAT() {
    return this.VAT;
}

public String getInvoice_Place() {
    return this.invoice_Place;
}

public String getInvoice_extD() {
    return this.invoice_extD;
}

public Vector[] getCancellation() {
    return this.cancellationFee;
}
```

```
    }

    public String getCurrency() {
        return this.Currency;
    }

    public String getBill_Top() {
        return this.Bill_Top;
    }

    public String getBill_Bottom() {
        return this.Bill_Bottom;
    }
}
```

Listing C.4 DB_Helpers class source code.

```
/*
 * This file is part of FGMP-Hotelverwaltung
 *
 * Copyright 2010, 2009 Daniel Fischer, David Gawehn, Martin Meyer, Christian
 * Pusch
 *
 * FGMP-Hotelverwaltung is free software: you can redistribute it and/or
 * modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package FGMP_Hotel_Management.Datenbank2;

import FGMP_Hotel_Management.Language.ErrorMessage;
import java.sql.*;
import java.util.ArrayList;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.DefaultComboBoxModel;
```

```

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import FGMP_Hotel_Management.Messaging;

/**
 * Provide several "helpers" for the DB access
 *
 * @author Daniel Fischer, David Gawehn
 */
public class DB_Helpers {

    /**
     * Provides the nearest free ID of a table
     * MAY STILL BE OPTIMIZED
     *
     * @param Table    DB-Table
     * @param Column  Column IDs
     * @return         -1, if failed, otherwise ID
     */
    public static int getNextID(String Table, String Column) {
        try {
            Statement stmt_id = DB_Backend.getConnection().createStatement();
            ResultSet max_id = stmt_id.executeQuery("SELECT MAX(" + Column +
                ") FROM " + Table);
            max_id.next();
            return max_id.getInt(1) + 1;
        } catch (SQLException ex) {
            Logger.getLogger(DB_Helpers.class.getName()).log(Level.SEVERE,
                null, ex);
            return -1;
        }
    }

    /**
     * Delete a record from a table
     *
     * @param Table    table Name
     * @param where    column Name
     * @param what     column entry
     * @return         -1, if failed, otherwise 1
     */
    public static int delEntry(String Table, String where, int what){
        try{
            PreparedStatement stmt_id =
                DB_Backend.getConnection().prepareStatement("DELETE FROM " +
                    Table + " WHERE " + where + " = ?");
            stmt_id.setInt(1,what);
        }
    }
}

```

```

        stmt_id.execute();
        return 1;
    } catch (SQLException ex){
        Logger.getLogger(DB_Helpers.class.getName()).log(Level.SEVERE,
            null, ex);
        Messaging.show_Dialog(ErrorMsg.msg[1], "Error",
            JOptionPane.ERROR_MESSAGE);
        return -1;
    }
}

public static int delEntry(String Table, String where, String what){
    try{
        PreparedStatement stmt_id =
            DB_Backend.getConnection().prepareStatement("DELETE FROM " +
                Table + " WHERE " + where + " = ?");
        stmt_id.setString(1,what);
        stmt_id.execute();
        return 1;
    } catch (SQLException ex){
        Logger.getLogger(DB_Helpers.class.getName()).log(Level.SEVERE,
            null, ex);
        Messaging.show_Dialog(ErrorMsg.msg[1], "Error",
            JOptionPane.ERROR_MESSAGE);
        return -1;
    }
}

/**
 * Fill a combo box with entries from the database conditions
 *
 * @param Model          DefaultComboBoxModel
 * @param List_id        ArrayList of entries to be filled
 * @param Table          DB-Table
 * @param Column_name    DB column names
 * @param Column_ID      DB column IDs
 */
public static void getComboItems(DefaultComboBoxModel Model, ArrayList
    List_id, String Table, String Column_name, String Column_ID) {
    try {
        if (Model != null) {
            Model.removeAllElements();
        }
        List_id.clear();
        Statement stmt = DB_Backend.getConnection().createStatement();

```

```

        ResultSet RS = stmt.executeQuery("SELECT * FROM " + Table + "
            ORDER BY " + Column_ID);
        while (RS.next()) {
            if (Model != null) {
                Model.addElement(RS.getString(Column_name));
            }
            List_id.add(RS.getString(Column_ID));
        }
    } catch (SQLException ex) {
        Logger.getLogger(DB_Helpers.class.getName()).log(Level.SEVERE,
            null, ex);
    }
}

/**
 * Fill a JList with entries from the database conditions
 *
 * @param ListModel      DefaultListModel
 * @param Liste          ArrayList of IDs
 * @param Table          DB-Table
 * @param Column_name    DB-Column names
 * @param Column_ID     DB-Column IDs
 */
public static void getListItems(DefaultListModel ListModel, ArrayList
    Liste, String Table, String Column_name, String Column_ID) {
    try {
        Statement stmt = DB_Backend.getConnection().createStatement();
        Liste.clear();
        ListModel.clear();
        ResultSet rs = stmt.executeQuery("SELECT * FROM " + Table);
        while (rs.next()) {
            ListModel.addElement(rs.getString(Column_name));
            if (Liste != null) {
                Liste.add(rs.getString(Column_ID));
            }
        }
    } catch (SQLException ex) {
        Logger.getLogger(DB_Helpers.class.getName()).log(Level.SEVERE,
            null, ex);
    }
}

/**
 * Checks whether a room loaded from the database can be deleted
 */
public static boolean isRoomDeletable(int ID) {
    try {

```

```

        PreparedStatement stmt1 =
            DB_Backend.getConnection().prepareStatement("SELECT * FROM
                booking_room WHERE room_id = ?");
        stmt1.setInt(1, ID);
        ResultSet rs1 = stmt1.executeQuery();
        while(rs1.next()) {
            int counter = 0;
            PreparedStatement stmt2 =
                DB_Backend.getConnection().prepareStatement("SELECT * FROM
                    booking WHERE booking_id= ? && paid = '0'");
            stmt2.setString(1, rs1.getString("booking_id"));
            ResultSet rs2 = stmt2.executeQuery();
            while(rs2.next()) {
                counter++;
            }
            if (counter > 0) {
                return false;
            }
        }
    }

} catch (SQLException ex) {
    Logger.getLogger(DB_Helpers.class.getName()).log(Level.SEVERE,
        null, ex);
}
return true;
}

/**
 * Returns the number rooms in use at the specified date d
 *
 * @param d date
 * @return number of rooms
 */

public static int getReservedRoomsAtDate(Date d) {
    int res = 0;
    try {
        PreparedStatement stmt1 =
            DB_Backend.getConnection().prepareStatement("SELECT * FROM
                booking,booking_room WHERE booking.arrivaldate <= ? AND
                booking.departuredate >= ? AND booking.cancellation = 0 AND
                booking.booking_id = booking_room.booking_id");
        stmt1.setDate(1, d);
        stmt1.setDate(2, d);
        ResultSet rs1 = stmt1.executeQuery();
        while(rs1.next()) {
            res++;
        }
    }
}

```



```

    }
    } catch (SQLException ex) {
        Logger.getLogger(DB_Helpers.class.getName()).log(Level.SEVERE,
            null, ex);
    }
    return res;
}

/**
 * Returns the number of rooms in the database
 *
 * @return number of rooms
 */

public static int getNumberOfRooms() {
    int rooms = 0;
    try {
        PreparedStatement stmt1 =
            DB_Backend.getConnection().prepareStatement("SELECT * FROM
                room");
        ResultSet rs1 = stmt1.executeQuery();
        while (rs1.next()) {
            rooms++;
        }
    } catch (SQLException ex) {
        Logger.getLogger(DB_Helpers.class.getName()).log(Level.SEVERE,
            null, ex);
    }
    return rooms;
}
}

```

Listing C.5 RelationSpouse class source code.

```

/*
 * FamilyTree - Family tree modeling software
 * created for research purposes
 * Copyright - Helsinki University of Technology,
 * Software Business and Engineering Institute
 * Created on 23.7.2003
 */
package familytree.model;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Iterator;

```

```
import javax.swing.JOptionPane;

public class RelationSpouse extends Relation {
    public Person husband;
    public Person wife;

    public RelationSpouse (Person husband, Person wife)
    {
        this.husband = husband;
        this.wife = wife;
    }

    public Person getHusband() {
        return husband;
    }

    public Person getWife() {
        return wife;
    }

    public Person getPerson1(){
        return getWife();
    }

    public Person getPerson2(){
        return getHusband();
    }

    public boolean equals(Object obj){
        if (obj instanceof RelationSpouse){
            RelationSpouse relation = (RelationSpouse) obj;
            return (this.getHusband().equals(relation.getHusband())&&
                this.getWife().equals(relation.getWife()));
        }
        return false;
    }

    public Person getPartner(Person person){
        if (person.equals(husband))
            return wife;
        else if (person.equals(wife))
            return husband;
        else{
            JOptionPane.showMessageDialog(null, "Wrong person type for spouse
                relationship...");
        }
    }
}
```

```

        return null;
    }
}

public String getRelationType(Person person){
    if (person.equals(husband))
        return WIFE;
    else if (person.equals(wife))
        return HUSBAND;
    else{
        JOptionPane.showMessageDialog(null, "Wrong person type for spouse
            relationship...");
        return INVALID_RELATIONSHIP;
    }
}

public void printToFile(File f) throws IOException{
    FileWriter fw = new FileWriter(f, false);
    fw.write("*** Outputting RelationSpouse ***\n");
    fw.write("Husband:
        "+husband.getFirstName()+husband.getLastName()+"("+husband.getId()+")\n");
    fw.write("Wife:
        "+wife.getFirstName()+wife.getLastName()+"("+wife.getId()+")\n");
    Iterator it = husband.getChildren().iterator();
    while (it.hasNext()) {
        Person child = (Person) it.next();
        if (child.getGender().equals(Person.GENDER_FEMALE))
            fw.write(" Daughter: "+child.getFirstName()+"\n");
        else if (child.getGender().equals(Person.GENDER_MALE))
            fw.write(" Son: "+child.getFirstName()+"\n");
    }
    fw.close();
}
}
}

```

Listing C.6 RelationParentChild class source code.

```

/*
 * FamilyTree - Family tree modeling software
 * created for research purposes
 * Copyright - Helsinki University of Technology,
 * Software Business and Engineering Institute
 * Created on 23.7.2003
 */
package familytree.model;

```

```
public class RelationParentChild extends Relation {
    private Person child;
    private Person parent;

    public RelationParentChild (Person child, Person parent){
        this.child = child;
        this.parent = parent;
    }

    public boolean isParent(Person person){
        return person == parent;
    }

    public boolean isChild(Person person){
        return person == child;
    }

    public Person getChild() {
        return child;
    }

    public Person getParent() {
        return parent;
    }

    public Person getPerson1(){
        return getParent();
    }

    public Person getPerson2(){
        return getChild();
    }

    public boolean equals(Object obj){
        if (obj instanceof RelationParentChild) {
            RelationParentChild relation = (RelationParentChild) obj;
            return (this.getChild().equals(relation.getChild()) &&
                this.getParent().equals(relation.getParent()));
        }
        return false;
    }

    public Person getPartner(Person person){
        if (person.equals(child))
            return parent;
        else if (person.equals(parent))
            return child;
    }
}
```

```
        else
            return null;
    }

    public String getRelationType(Person person) {
        if (person.equals(child)) {
            if (parent.isFemale())
                return MOTHER;
            else
                return FATHER;
        } else if (person.equals(parent)) {
            if (child.isFemale())
                return DAUGHTER;
            else
                return SON;
        } else
            return INVALID_RELATIONSHIP;
    }
}
```


Appendix

D

**DATA TRANSFORMATION AND APPLICATION OF
FLEISS KAPPA TEST**

Listing D.1 R functions created to support necessary transformations.

```

convertRatings <- function (ratings)
{
  lev <- levels(as.factor(ratings))
  ratings <- as.matrix(na.omit(ratings))
  ns <- nrow(ratings)
  target <- array(0, dim=c(ns,4))
  for (i in 1:ns) {
    if(ratings[i] == lev[1])
      target[i, 1] <- 1
    else if(ratings[i] == lev[2])
      target[i, 2] <- 1
    else if(ratings[i] == lev[3])
      target[i, 3] <- 1
    else if(ratings[i] == lev[4])
      target[i, 4] <- 1
  }
  return(target)
}

convertRatings3 <- function (ratings)
{
  #ns is used just to iterate the for loop
  # and to define the number of columns in the target matrix
  ns <- nrow(ratings)

  # the resulting matrix
  target <- array(dim=c(2,ns))

  for (i in 1:ns){
    for (j in 1:4) {
      if ((j == 1) && (ratings[i, j] == 1)){
        target[1,i] = ">"
        target[2,i] = "<"
        j = 5
      }
      else if ((j == 2) && (ratings[i, j] == 1)){
        target[1,i] = "<"
        target[2,i] = ">"
        j = 5
      }
      else if ((j == 3) && (ratings[i, j] == 1)){
        target[1,i] = "="
        target[2,i] = "="
        j = 5
      }
    }
  }
}

```



```
    }  
  }  
  return(target)  
}
```

Listing D.2 Fleiss Kappa in R and results.

```
# loading IRR library package  
library(irr)  
  
# loading R syntax file for the survey data  
source("survey_44763_R_syntax_file.R", encoding = "UTF-8")  
  
# Transformation 1 for the ratings  
ratings1 <- convertRatings (data[,19])  
ratings2 <- convertRatings (data[,21])  
ratings3 <- convertRatings (data[,23])  
  
# Transformation 2 for the ratings  
ratings1_1 <- convertRatings3(ratings1)  
ratings2_1 <- convertRatings3(ratings2)  
ratings3_1 <- convertRatings3(ratings3)  
  
# Removing rating 52 which corresponds to "I don't know" answer.  
# There might be a better way to do it.  
ratings1_1_NAomited <- ratings1_1[,-52]  
ratings2_1_NAomited <- ratings2_1[,-52]  
ratings3_1_NAomited <- ratings3_1[,-41]  
  
# Executing Fleiss Kappa test and corresponding output  
kappam.fleiss(ratings1_1_NAomited)  
> Fleiss' Kappa for m Raters  
>  
> Subjects = 2  
> Raters = 79  
> Kappa = 0.193  
>  
> z = 21.4  
> p-value = 0  
  
kappam.fleiss(ratings2_1_NAomited)  
> Fleiss' Kappa for m Raters  
>  
> Subjects = 2  
> Raters = 79  
> Kappa = 0.234  
>
```

```
>      z = 25.1
> p-value = 0

kappam.fleiss(ratings3_1_NAomited)
> Fleiss' Kappa for m Raters
>
> Subjects = 2
> Raters = 79
> Kappa = 0.155
>
>      z = 17.1
> p-value = 0
```

Appendix

E

APPLICATION OF FISHER EXACT TEST

Listing E.1 Fisher exact test for cohesion familiarity vs. cohesion ratings.

```

# http://stat.ethz.ch/R-manual/R-patched/library/stats/html/fisher.test.html
# Creating matrix: cohesion familiarity vs. ratings

> cohFam_Ratings1 <- matrix(c(41,4,26,0, 4,0,4,1), nrow = 2, ncol = 4, byrow =
  = TRUE, dimnames = list(c("Yes", "No"), c("1st more cohesive", "2nd more
  cohesive", "Equally coh", "Don't know")))

> cohFam_Ratings1
  1st more cohesive 2nd more cohesive Equally coh Don't know
Yes                41                4                26                0
No                 4                 0                 4                 1

> cohFam_Ratings2 <- matrix(c(50,9,12,0, 4,1,3,1), nrow = 2, ncol = 4, byrow =
  TRUE, dimnames = list(c("Yes", "No"), c("1st more cohesive", "2nd more
  cohesive", "Equally coh", "Don't know")))

> cohFam_Ratings2
  1st more cohesive 2nd more cohesive Equally coh Don't know
Yes                50                9                12                0
No                 4                 1                 3                 1

> cohFam_Ratings3 <- matrix(c(5,41,24,1, 1,2,6,0), nrow = 2, ncol = 4, byrow =
  TRUE, dimnames = list(c("Yes", "No"), c("1st more cohesive", "2nd more
  cohesive", "Equally coh", "Don't know")))

> cohFam_Ratings3
  1st more cohesive 2nd more cohesive Equally coh Don't know
Yes                5                41                24                1
No                 1                 2                 6                 0

# Executing the Fisher exact test between Cohesion familiarity and Ratings

> fisher.test(cohFam_Ratings1, simulate.p.value = TRUE)

      Fisher's Exact Test for Count Data with simulated p-value (based on
      2000 replicates)

data: cohFam_Ratings1
p-value = 0.1704
alternative hypothesis: two.sided

> fisher.test(cohFam_Ratings1)

```

```

    Fisher's Exact Test for Count Data

data: cohFam_Ratings1
p-value = 0.1753
alternative hypothesis: two.sided

> fisher.test(cohFam_Ratings2, simulate.p.value = TRUE)

    Fisher's Exact Test for Count Data with simulated p-value (based on
    2000 replicates)

data: cohFam_Ratings2
p-value = 0.05447
alternative hypothesis: two.sided

> fisher.test(cohFam_Ratings2)

    Fisher's Exact Test for Count Data

data: cohFam_Ratings2
p-value = 0.06224
alternative hypothesis: two.sided

> fisher.test(cohFam_Ratings3, simulate.p.value = TRUE)

    Fisher's Exact Test for Count Data with simulated p-value (based on
    2000 replicates)

data: cohFam_Ratings3
p-value = 0.1544
alternative hypothesis: two.sided

> fisher.test(cohFam_Ratings3)

    Fisher's Exact Test for Count Data

data: cohFam_Ratings3
p-value = 0.1523
alternative hypothesis: two.sided

```

Listing E.2 Fisher exact test for programming experience (years) vs. cohesion ratings (of each scenario).

```

*****
Three categories:

```

```

* 1st category (<=5 years): 1st (lower) quartile;
* 2nd category (>5 and > 12.25 years): inter quartile range;
* 3rd category (>=12.25 years): 3rd (upper) quartile;
*****

# Creating matrix: years of programming experience vs. ratings

ratings1_1_YearsProg <- matrix(c(10,21,14, 1,2,1, 10,15,5, 0,1,0), nrow = 4,
  ncol = 3, byrow = TRUE, dimnames = list(c("1st more cohesive", "2nd more
  cohesive", "Equally coh", "Don't know"), c("<=5", ">5 AND <12.25",
  ">=12.25")))

> ratings1_1_YearsProg
      <=5 >5 AND <12.25 >=12.25
1st more cohesive  10          21      14
2nd more \ncohesive 1           2       1
Equally coh       10          15       5
Don't know        0           1       0
>

ratings2_1_YearsProg <- matrix(c(14,24,16, 3,5,2, 4,9,2, 0,1,0), nrow = 4,
  ncol = 3, byrow = TRUE, dimnames = list(c("1st more cohesive", "2nd more
  cohesive", "Equally coh", "Don't know"), c("<=5", ">5 AND <12.25",
  ">=12.25")))

> ratings2_1_YearsProg
      <=5 >5 AND <12.25 >=12.25
1st more cohesive  14          24      16
2nd more cohesive  3           5       2
Equally coh        4           9       2
Don't know         0           1       0
>

ratings3_1_YearsProg <- matrix(c(2,4,0, 10,19,14, 9,15,6, 0,1,0), nrow = 4,
  ncol = 3, byrow = TRUE, dimnames = list(c("1st more cohesive", "2nd more
  cohesive", "Equally coh", "Don't know"), c("<=5", ">5 AND <12.25",
  ">=12.25")))

> ratings3_1_YearsProg
      <=5 >5 AND <12.25 >=12.25
1st more cohesive   2           4       0
2nd more \ncohesive 10          19      14
Equally coh         9           15       6
Don't know          0           1       0

```

```
# Executing the Fisher exact test between Years of Prog. Exp. and Ratings
> fisher.test(ratings1_1_YearsProg, simulate.p.value = TRUE)

      Fisher's Exact Test for Count Data with simulated p-value (based on
      2000 replicates)

data: ratings1_1_YearsProg
p-value = 0.7716
alternative hypothesis: two.sided

> fisher.test(ratings1_1_YearsProg)

      Fisher's Exact Test for Count Data

data: ratings1_1_YearsProg
p-value = 0.7688
alternative hypothesis: two.sided

>

> fisher.test(ratings2_1_YearsProg, simulate.p.value = TRUE)

      Fisher's Exact Test for Count Data with simulated p-value (based on
      2000 replicates)

data: ratings2_1_YearsProg
p-value = 0.8851
alternative hypothesis: two.sided

>

> fisher.test(ratings2_1_YearsProg)

      Fisher's Exact Test for Count Data

data: ratings2_1_YearsProg
p-value = 0.8749
alternative hypothesis: two.sided

>
```

```

> fisher.test(ratings3_1_YearsProg, simulate.p.value = TRUE)

      Fisher's Exact Test for Count Data with simulated p-value (based on
      2000 replicates)

data: ratings3_1_YearsProg
p-value = 0.5867
alternative hypothesis: two.sided

> fisher.test(ratings3_1_YearsProg)

      Fisher's Exact Test for Count Data

data: ratings3_1_YearsProg
p-value = 0.5863
alternative hypothesis: two.sided

>

```

Listing E.3 Fisher exact test for cohesion familiarity vs. academic degree.

```

# http://stat.ethz.ch/R-manual/R-patched/library/stats/html/fisher.test.html
# Creating matrix: academic degree vs. cohesion familiarity

> cohFam_AcadDegree <- matrix(c(2,20,10,31,8,0, 0,5,1,2,1,0), nrow = 2, ncol =
  6, byrow = TRUE, dimnames = list(c("Yes", "No"), c("Undergrad", "Grad",
  "Grad Cert Expert", "Master", "PhD", "No degree")))
> cohFam_AcadDegree
  Undergrad Grad Grad Cert Expert Master PhD No degree
Yes      2  20          10  31  8      0
No       0   5           1   2  1      0

# Executing the Fisher exact test between Academic degree and Cohesion
  Familiarity

> fisher.test(cohFam_AcadDegree, simulate.p.value = TRUE)

      Fisher's Exact Test for Count Data with simulated p-value (based on
      2000 replicates)

data: cohFam_AcadDegree
p-value = 0.5512
alternative hypothesis: two.sided

> fisher.test(cohFam_AcadDegree)

```


Fisher's Exact Test for Count Data

```
data: cohFam_AcadDegree
p-value = 0.5464
alternative hypothesis: two.sided
```

Listing E.4 Fisher exact test for Academic degree vs. cohesion ratings (of each scenario).

```
# http://stat.ethz.ch/R-manual/R-patched/library/stats/html/fisher.test.html

# Creating matrix: academic degree vs. ratings

> ratings1_AcadDegree <- matrix(c(1,12,5,21,6,0, 0,2,0,2,1,0, 1,10,6,10,3,0,
  0,1,0,0,0,0), nrow = 4, ncol = 6, byrow = TRUE, dimnames = list(c("1st
  more cohesive", "2nd more cohesive", "Equally coh", "Don't know"),
  c("Undergrad", "Grad", "Grad Cert Expert", "Master", "PhD", "No degree")))

> ratings1_AcadDegree
      Undergrad Grad Grad Cert Expert Master PhD No degree
1st more cohesive      1  12          5   21  6      0
2nd more cohesive      0   2          0    2  1      0
Equally coh            1  10          6   10  3      0
Don't know             0   1          0    0  0      0

> ratings2_AcadDegree <- matrix(c(2,17,9,22,5,0, 0,3,2,1,4,0, 0,4,0,10,0,0,
  0,1,0,0,0,0), nrow = 4, ncol = 6, byrow = TRUE, dimnames = list(c("1st
  more cohesive", "2nd more cohesive", "Equally coh", "Don't know"),
  c("Undergrad", "Grad", "Grad Cert Expert", "Master", "PhD", "No degree")))

> ratings2_AcadDegree
      Undergrad Grad Grad Cert Expert Master PhD No degree
1st more cohesive      2  17          9   22  5      0
2nd more cohesive      0   3          2    1  4      0
Equally coh            0   4          0   10  0      0
Don't know             0   1          0    0  0      0

> ratings3_AcadDegree <- matrix(c(0,1,1,3,1,0, 1,15,5,17,5,0, 1,9,4,13,3,0,
  0,0,1,0,0,0), nrow = 4, ncol = 6, byrow = TRUE, dimnames = list(c("1st
  more cohesive", "2nd more cohesive", "Equally coh", "Don't know"),
  c("Undergrad", "Grad", "Grad Cert Expert", "Master", "PhD", "No degree")))

> ratings3_AcadDegree
      Undergrad Grad Grad \nCert Expert Master PhD No degree
1st more cohesive      0   1          1    3  1      0
```

2nd more cohesive	1	15	5	17	5	0
Equally coh	1	9	4	13	3	0
Don't know	0	0	1	0	0	0

```
# Executing the Fisher exact test between Academic degree and Ratings
```

```
> fisher.test(ratings1_AcadDegree, simulate.p.value = TRUE)
```

```
Fisher's Exact Test for Count Data with simulated p-value (based on  
2000 replicates)
```

```
data: ratings1_AcadDegree  
p-value = 0.8656  
alternative hypothesis: two.sided
```

```
> fisher.test(ratings1_AcadDegree)
```

```
Fisher's Exact Test for Count Data
```

```
data: ratings1_AcadDegree  
p-value = 0.8646  
alternative hypothesis: two.sided
```

```
> fisher.test(ratings2_AcadDegree, simulate.p.value = TRUE)
```

```
Fisher's Exact Test for Count Data with simulated p-value (based on  
2000 replicates)
```

```
data: ratings2_AcadDegree  
p-value = 0.04098  
alternative hypothesis: two.sided
```

```
> fisher.test(ratings2_AcadDegree)
```

```
Fisher's Exact Test for Count Data
```

```
data: ratings2_AcadDegree  
p-value = 0.03684  
alternative hypothesis: two.sided
```

```
> fisher.test(ratings3_AcadDegree, simulate.p.value = TRUE)
```

```
Fisher's Exact Test for Count Data with simulated p-value (based on  
2000 replicates)
```

```
data: ratings3_AcadDegree
p-value = 0.8716
alternative hypothesis: two.sided
```

```
> fisher.test(ratings3_AcadDegree)
```

```
      Fisher's Exact Test for Count Data
```

```
data: ratings3_AcadDegree
p-value = 0.8831
alternative hypothesis: two.sided
```


Appendix

F

**R SCRIPTS FOR CORRELATION TESTS AND
REGRESSION TREES**

Listing F.1 R script for generating and plotting regression trees.

```

library(rpart)
library(rpart.plot)

TOMCAT <- read.table("cohesionMetricsFinal_andCC_Tomcat_2014.csv",
  header=TRUE, sep=",")
attach(TOMCAT)
tomcat_fit <- rpart(CC ~ LCOM2 + LCOM3 + LCOM4 + LCOM5 + TCC + LCbC_XScan +
  MWE)
prp(tomcat_fit, main="Tomcat - Change Count Regression Tree", varlen=10,
  faclen = 0, cex = 0.7, extra = 101, type=1)
detach(TOMCAT)

FINDBUGS <- read.table("cohesionMetricsFinal_andCC_Findbugs_2014.csv",
  header=TRUE, sep=",")
attach(FINDBUGS)
findbugs_fit <- rpart(CC ~ LCOM2 + LCOM3 + LCOM4 + LCOM5 + TCC + LCbC_XScan +
  MWE)
prp(findbugs_fit, main="Findbugs - Change Count Regression Tree", varlen=10,
  faclen = 0, cex = 0.7, extra = 101, type=1)
detach(FINDBUGS)

FREECOL <- read.table("cohesionMetricsFinal_andCC_Freecol_2014.csv",
  header=TRUE, sep=",")
attach(FREECOL)
freecol_fit <- rpart(CC ~ LCOM2 + LCOM3 + LCOM4 + LCOM5 + TCC + LCbC_XScan +
  MWE)
prp(freecol_fit, main="Freecol - Change Count Regression Tree", varlen=10,
  faclen = 0, cex = 0.7, extra = 101, type=1)
detach(FREECOL)

JFREECHART <- read.table("cohesionMetricsFinal_andCC_JFreeChart_2014.csv",
  header=TRUE, sep=",")
attach(JFREECHART)
jfreechart_fit <- rpart(CC ~ LCOM2 + LCOM3 + LCOM4 + LCOM5 + TCC + LCbC_XScan
  + MWE)
prp(jfreechart_fit, main="JFreeChart - Change Count Regression Tree",
  varlen=10, faclen = 0, cex = 0.7, extra = 101, type=1)
detach(JFREECHART)

RHINO <- read.table("cohesionMetricsFinal_andCC_Rhino_1.6R5.csv",
  header=TRUE, sep=",")
attach(RHINO)
rhino_fit <- rpart(CC ~ LCOM2 + LCOM3 + LCOM4 + LCOM5 + TCC + LCbC_XScan +
  MWE)
prp(rhino_fit, main="Rhino - Change Count Regression Tree", varlen=10, faclen

```

```

    = 0, cex = 0.7, extra = 101, type=1)
detach(RHINO)

JEDIT <- read.table("cohesionMetricsFinal_andCC_jEdit_4.3.2.csv",
  header=TRUE, sep=",")
attach(JEDIT)
jedit_fit <- rpart(CC ~ LCOM2 + LCOM3 + LCOM4 + LCOM5 + TCC + LCbC_XScan +
  MWE)
prp(jedit_fit, main="jEdit - Change Count Regression Tree", varlen=10, faclen
  = 0, cex = 0.7, extra = 101, type=1)
detach(JEDIT)

```

Listing F.2 R script for executing Spearman rank correlation test.

```

library(Hmisc) # library for 'rcorr()' function

TOMCAT <- read.table("cohesionMetricsFinal_andCC_Tomcat_2014.csv",
  header=TRUE, sep=",") # reading the data
rcorr(as.matrix(TOMCAT), type="spearman") # running correlation (it considers
  pairwise deletion for missing values)

RHINO <- read.table("cohesionMetricsFinal_andCC_Rhino_1.6R5.csv",
  header=TRUE, sep=",")
rcorr(as.matrix(RHINO), type="spearman")

JFREECHART <- read.table("cohesionMetricsFinal_andCC_JFreeChart_2014.csv",
  header=TRUE, sep=",")
rcorr(as.matrix(JFREECHART), type="spearman")

JEDIT <- read.table("cohesionMetricsFinal_andCC_jEdit_4.3.2.csv",
  header=TRUE, sep=",")
rcorr(as.matrix(JEDIT), type="spearman")

FINDBUGS <- read.table("cohesionMetricsFinal_andCC_Findbugs_2014.csv",
  header=TRUE, sep=",")
rcorr(as.matrix(FINDBUGS), type="spearman")

FREECOL <- read.table("cohesionMetricsFinal_andCC_Freecol_2014.csv",
  header=TRUE, sep=",")
rcorr(as.matrix(FREECOL), type="spearman")

```




REGRESSION TREES

Tomcat - Change Count Regression Tree

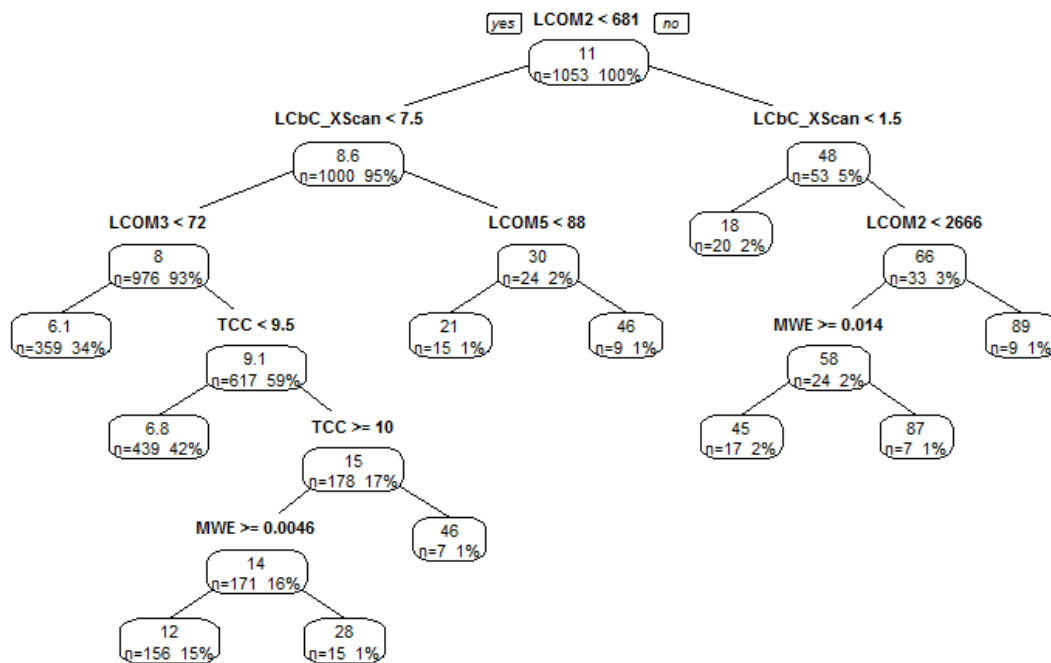


Figure G.1 Tomcat regression tree.

Findbugs - Change Count Regression Tree

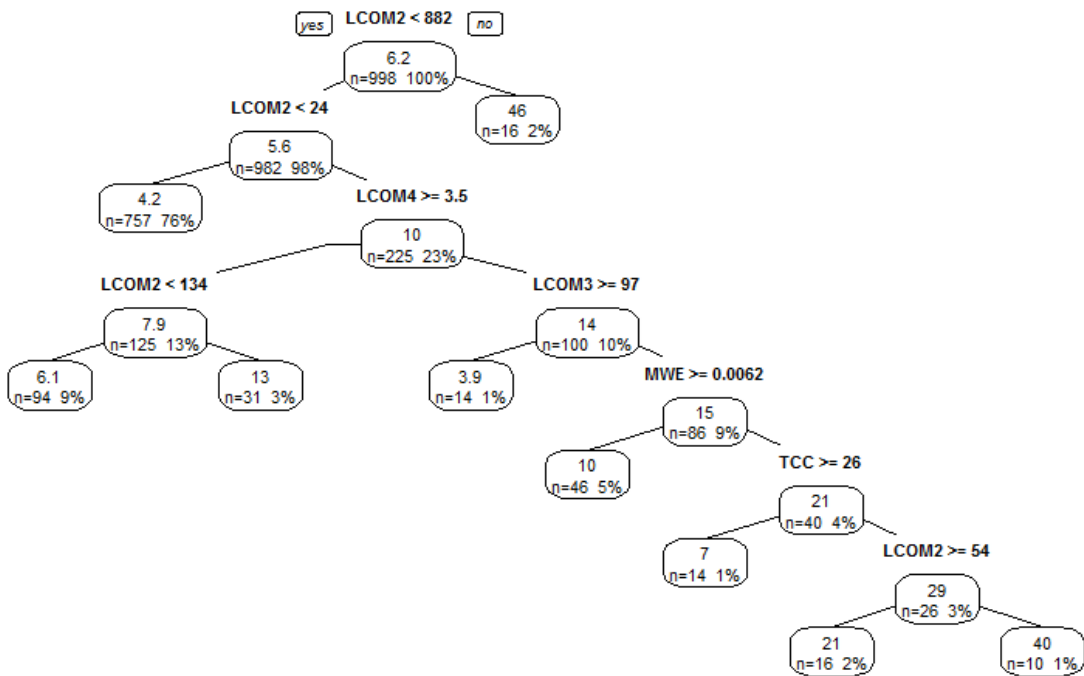


Figure G.2 Findbugs regression tree.

Freecol - Change Count Regression Tree

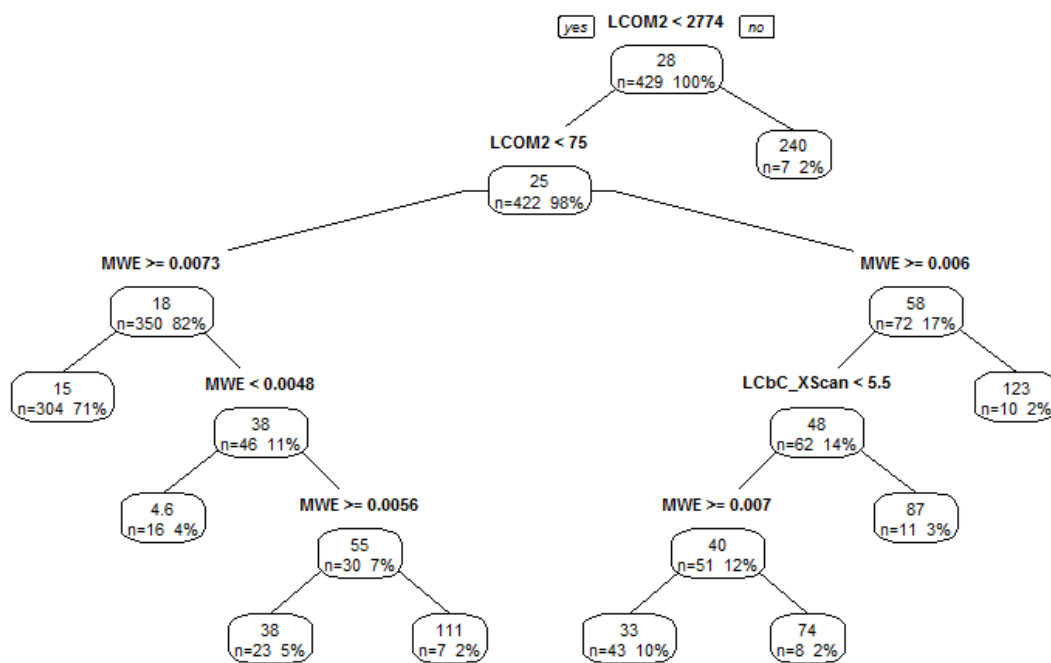


Figure G.3 Freecol regression tree.

JFreeChart - Change Count Regression Tree

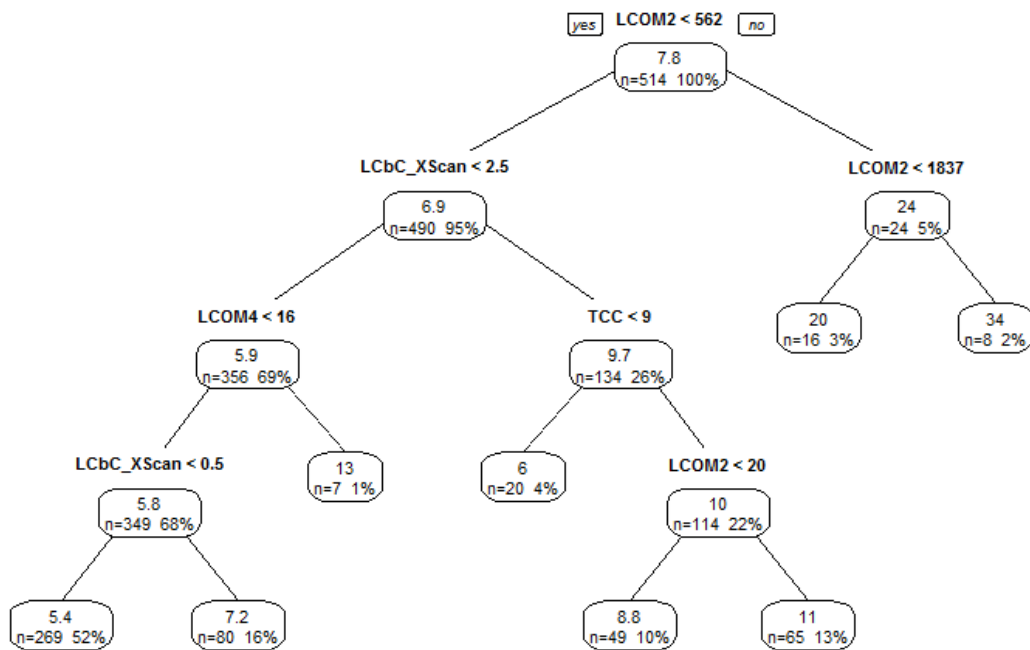


Figure G.4 JFreeChart regression tree.

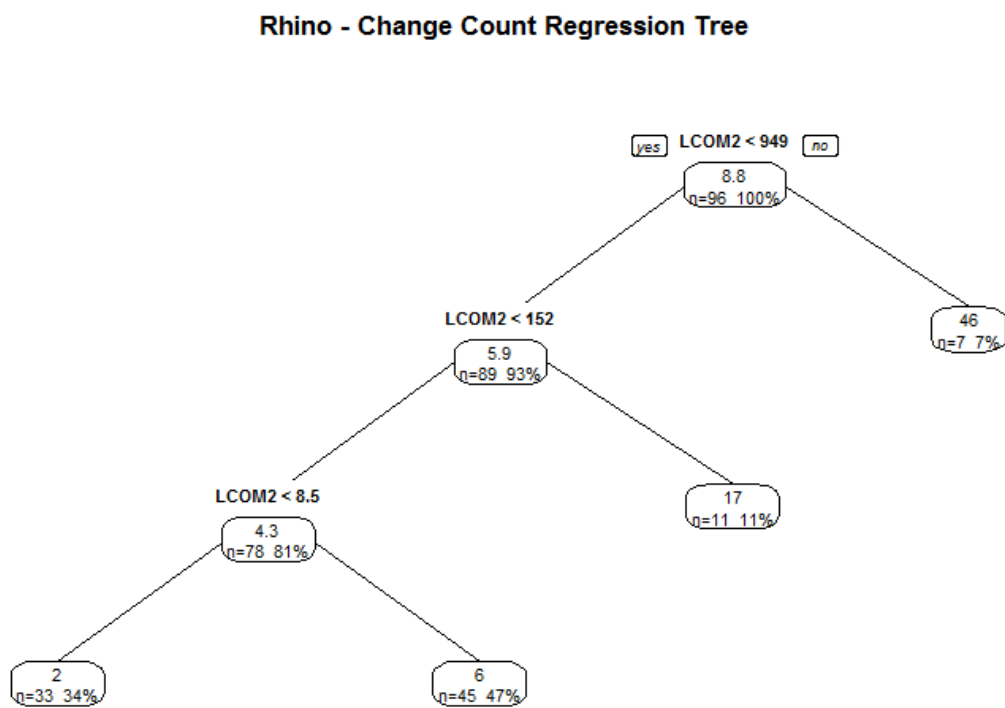


Figure G.5 Rhino regression tree.

jEdit - Change Count Regression Tree

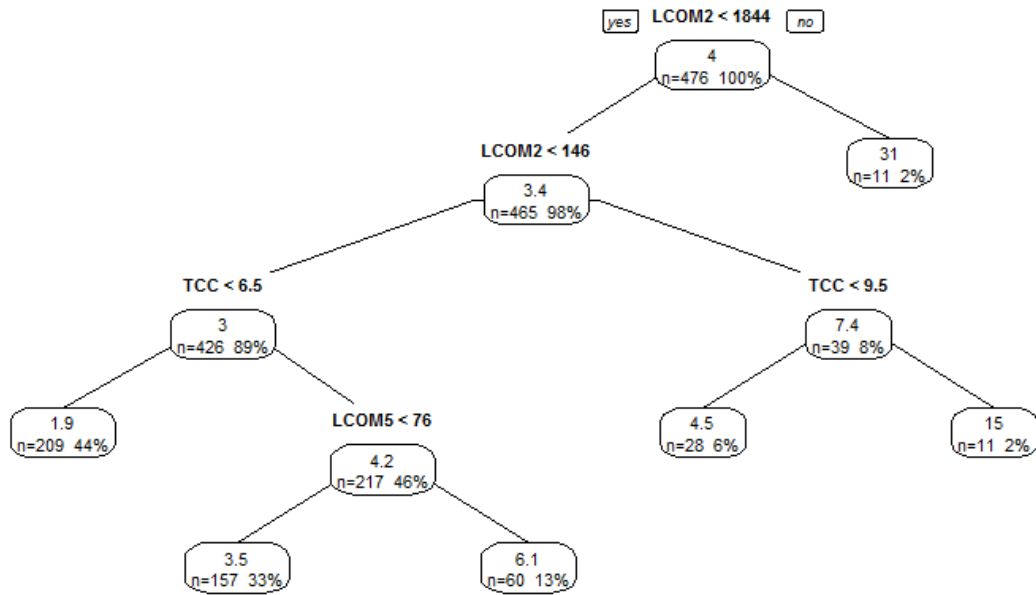


Figure G.6 JEdit regression tree.

Appendix

H

APPLICATION OF FRIEDMAN TEST

Listing H.1 R script and log for executing Friedman test.

```
# It is necessary to load the LCbC columns for each system beforehand.

# Tomcat, JFreeChart, Freecol and Findbugs: only two columns (LCbC_XScan vs.
  LCbC_Topics)
> friedman.test(as.matrix(Tomcat_LCbC_Topics_XScan[,2:3]))

      Friedman rank sum test

data: as.matrix(Tomcat_LCbC_Topics_XScan[, 2:3])
Friedman chi-squared = 367.0662, df = 1, p-value < 2.2e-16

> friedman.test(as.matrix(JFreeChart_LCbC_Topics_XScan[,2:3]))

      Friedman rank sum test

data: as.matrix(JFreeChart_LCbC_Topics_XScan[, 2:3])
Friedman chi-squared = 53.5078, df = 1, p-value = 2.576e-13

> friedman.test(as.matrix(Freecol_LCbC_Topics_XScan[,2:3]))

      Friedman rank sum test

data: as.matrix(Freecol_LCbC_Topics_XScan[, 2:3])
Friedman chi-squared = 68.3108, df = 1, p-value < 2.2e-16

> friedman.test(as.matrix(Findbugs_LCbC_Topics_XScan[,2:3]))

      Friedman rank sum test

data: as.matrix(Findbugs_LCbC_Topics_XScan[, 2:3])
Friedman chi-squared = 381.3333, df = 1, p-value < 2.2e-16

# jEdit and Rhino: three columns (all three LCbC variations)
> friedman.test(as.matrix(JEDIT))

      Friedman rank sum test

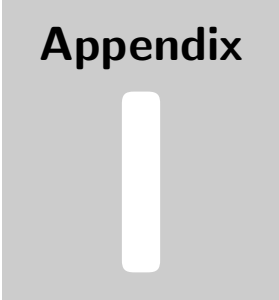
data: as.matrix(JEDIT)
Friedman chi-squared = 249.0053, df = 2, p-value < 2.2e-16
```

```
> friedman.test(as.matrix(RHINO))
```

```
    Friedman rank sum test
```

```
data: as.matrix(RHINO)
```

```
Friedman chi-squared = 154.7427, df = 2, p-value < 2.2e-16
```

JEDIT SET OF CONCERNS

UNDO: Concern that reverses the most recent editing command.

See more at <http://www.jedit.org/users-guide/undo-redo.html>.

REDO – Concern that restores editing changes.

See more: (<http://www.jedit.org/users-guide/undo-redo.fhtml>).

TRANSFERRING TEXT - Set of commands and actions for moving and copying text (including cut, paste and copy).

See more: (<http://www.jedit.org/users-guide/text-transfer.html>).

KEYBOARDS COMMANDS - For manipulating entire words, lines and paragraphs at a time.

MARKERS – It is a pointer to a specific location within a buffer, which may or may not have a single-character shortcut associated with it.

See more: (<http://www.jedit.org/users-guide/markers.html>).

SPLIT WINDOWS – A feature to split a window (view) into multiple panes.

See more: (<http://www.jedit.org/users-guide/views.html>)

MULTIPLE SELECTION - In multiple selection mode, multiple fragments of text can be selected and operated on simultaneously, and the caret can be moved independently of the selection.

See more: (<http://www.jedit.org/users-guide/selection.html#multi-select>).

WORD WRAP - Splits lines at word boundaries in order to fit text within a specified wrap margin.

See more: (<http://www.jedit.org/users-guide/word-wrap.html>).

SYNTAX HIGHLIGHTING – It is the display of programming language tokens using different fonts and colors.

See more: (<http://www.jedit.org/users-guide/modes.html#syntax-hilite>).

BRACKET MATCHING - jEdit has several features to make brackets easier to deal with.

See more: (<http://www.jedit.org/users-guide/bracket-matching.html>).

INDENTATION - jEdit makes a distinction between the tab width, which is used when displaying hard tab characters, and the indent width, which is used when a level of indent is to be added or removed, for example by mode-specific auto indent routines.

See more: (<http://www.jedit.org/users-guide/indent.html>).

CODE COMMENTS - jEdit has commands which make inserting comments more convenient.

See more: (<http://www.jedit.org/users-guide/commenting.html>).

ABBREVIATIONS - Abbreviations are invoked by typing a couple of letters and take the word before the caret as the abbreviation name.

See more: (<http://www.jedit.org/users-guide/abbrevs.html>).

FOLDING - Lets you selectively hide and show these sections, replacing hidden ones with a single line that serves as an “overview” of that section.

See more: (<http://www.jedit.org/users-guide/folding.html>).

SEARCH AND REPLACE – This Concern displays a dialog box that allows you search and replace text.

See more:(<http://www.jedit.org/users-guide/search-replace.html>).

OPEN FILE – This Concern displays a file system browser dialog box and loads the specified file into a new buffer.

See more: (<http://www.jedit.org/users-guide/opening.html>).

SAVE FILE – This Concern saves the current buffer to disk. It still allows rename the buffer and saves it in a new location(Save as) or It allows to save the buffer to a different location but does not rename the buffer (Save a Copy As).

See more: (<http://www.jedit.org/users-guide/saving.html>).

CLOSE FILE – This Concern closes the current buffer or all buffers (Close All).

See more: (<http://www.jedit.org/users-guide/closing-exiting.html>).

NEW FILE – This function opens a new, empty, buffer.

See more: (<http://www.jedit.org/users-guide/creating.html>).

CHARACTER ENCODING - A character encoding is a mapping from a set of characters to their on-disk representation.

See more: (<http://www.jedit.org/users-guide/encodings.html>).

PRINTING - This Concern prints the current buffer.

See more: (<http://www.jedit.org/users-guide/printing.html>).

TEXT INSERTION – It represents the text insertion (typing) in the text area.

See more: (<http://www.jedit.org/users-guide/entering-text.html>)

TEXT DELETION – It represents the text deletion in the text area.

See more: (<http://www.jedit.org/users-guide/entering-text.html>).

KEYBOARD FOCUS – Feature to represent where the keyboard typing is focusing in the screen.To ensure that the keyboard is focused in the textarea, you can always use

the mouse and click in it, but a more keyboard-friendly way is preferred when you are just about to start typing anyway. For this reason, a number of jEdit's actions have a side-effect of focusing on the text area as well. Ex: View - Toggle Full Screen; View - Scrolling - Scroll and Center Caret; View - Scrolling - Scroll to Line; View - Docking - Toggle Docked Areas; View - Docking - Close current docking area.

See more: (<http://www.jedit.org/users-guide/keyboard-focus.html>).

SCROLLING - If you have a mouse with a scroll wheel, you can use the wheel to scroll up and down in the text area. Keyboard commands for scrolling the text area are also available.

See more: (<http://www.jedit.org/users-guide/scrolling.html>).

MACROS RECORDING - Macros in jEdit are short scripts written in a scripting language called BeanShell. The simplest use of macros is to record a series of key strokes and menu commands as a BeanShell script, and play them back later.

See more: (<http://www.jedit.org/users-guide/recording-macros.html>).

MACROS RUNNING - For running an existing macro.

See more: (<http://www.jedit.org/users-guide/running-macros.html>).

PLUGIN MANAGER - This Concern displays the plugin manager window. It consists of three tabs: Manage, Update and Install.

See more: (<http://www.jedit.org/users-guide/plugin-manager.html>).

TIPS OF THE DAY - When starting jEdit, this function displays tips to help use it.

RELOADING FROM DISK - It used to reload the current buffer from disk at any other time. It still discards unsaved changes in all open buffers and reload them from disk (Reload All).

See more: (<http://www.jedit.org/users-guide/reloading.html>).

WINDOW DOCKING - A docking layout is similar to an Eclipse "Perspective" in that it describes a set of dockable windows that are visible to the user at any given time, hiding the rest.

See more: (<http://www.jedit.org/users-guide/docking.html>).

BEANSHELL (BSH) INTEGRATION - BeanShell is a small, free, embeddable, Java source interpreter with object scripting language Concerns, written in Java. BeanShell executes standard Java statements and expressions, in addition to obvious scripting commands and syntax. BeanShell supports scripted objects as simple method closures like those in Perl and JavaScript.

See more: (<http://www.jedit.org/users-guide/macro-tips-BeanShell.html>), (<http://www.jedit.org/users-guide/bsh-commands.html>).

BUFFER OPTIONS - It displays a dialog box for changing editor settings on a per-buffer basis.

See more: (<http://www.jedit.org/users-guide/buffer-opts.html>).

GLOBAL OPTIONS – This Concern display a dialog box that is divided into several panes, each pane containing a set of related options.

See more: (<http://www.jedit.org/users-guide/global-opts.html>).

PLUGIN OPTIONS – This Concern displays a dialog box for changing plugin settings.

See more: (<http://www.jedit.org/users-guide/plugin-manager.html>).

ACTION BAR - The action bar allows almost any editor Concern to be accessed from the keyboard.

See more: (<http://www.jedit.org/users-guide/action-bar.html>).

STATUS BAR – The status bar consists of a number of components, such as: caret position informatio, a message area where various prompts and status messages are shown etc.

See more: (<http://www.jedit.org/users-guide/status-bar.html>).

TOOL BAR - The file system browser has a tool bar containing a number of buttons. See more: (<http://www.jedit.org/users-guide/vfs-browser.html#d0e1964>).

SWITCHING BUFFER - Each EditPane has an optional drop-down BufferSwitcher at the top. The BufferSwitcher shows the current buffer and can also be used to switch the current buffer, using menu item commands and their keyboard shortcuts.

See more: (<http://www.jedit.org/users-guide/buffers.html>)

BUFFERSETS - The buffer sets Concern helps keep the buffer lists local and manageable when using jEdit in a multiple-View and multiple-EditPane environment.

See more: (<http://www.jedit.org/users-guide/buffersets.html>)

PLUGINS MENU – This menu displays plugins commands.

COMMANDS MENU – This menu displays commands like 'Parent Directory' and 'Reload Directory'.

FAVORITE MENU – This menu displays all files and directories in the favorites list.

EXITING JEdit – This Concern allows a completely exit jEdit, prompting if unsaved buffers should be saved first. See more: (<http://www.jedit.org/users-guide/closing-exiting.html>).

RANGE SELECTION - Dragging the mouse creates a range selection from where the mouse was pressed to where it was released.

See more: (<http://www.jedit.org/users-guide/selection.html#d0e2421>).

RECTANGULAR SELECTION – Selection of a text fragment by dragging the mouse with the control key held down for creating a rectangular selection.

See more: (<http://www.jedit.org/users-guide/selection.html#d0e2421>).

USEFUL RESOURCES:

- jEdit users guide:
<http://www.jedit.org/users-guide/index.html>;
<http://www.jedit.org/index.php?page=download>.
- jEdit features:
<http://www.jedit.org/index.php?page=features>.
- jEdit source code compiled:
<http://java.labsoft.dcc.ufmg.br/qualitas.class/download.html>