A Precise Method-Method Interaction-Based Cohesion Metric for Object-Oriented Classes

Jehad Al Dallal
Department of Information Science
Kuwait University
P.O. Box 5969, Safat 13060, Kuwait
j.aldallal@ku.edu.kw

Lionel C. Briand
Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway
Department of Informatics, University of Oslo, Norway
briand@simula.no

Abstract
The building of highly cohesive classes is an important objective in object-oriented design. Class cohesion refers to the relatedness of the class members, and it indicates one important aspect of the class design quality. A meaningful class cohesion metric helps object-oriented software developers detect class design weaknesses and refactor classes accordingly. Several class cohesion metrics have been proposed in the literature. Most of these metrics are applicable based on low-level design information such as attribute references in methods. Some of these metrics capture class cohesion by counting the number of method pairs sharing common attributes. A few metrics measure cohesion more precisely by considering the degree of interaction, through attribute references, between each pair of methods. However, the formulas applied by these metrics to measure the degree of interaction cause the metrics to violate important mathematical properties, thus undermining their construct validity and leading to misleading cohesion measurement. In this paper, we propose a formula that precisely measures the degree of interaction between each pair of methods, and we use it as a basis to introduce a low-level design class cohesion metric (LSCC). We verify that the proposed formula does not cause the metric to violate important mathematical properties. In addition, we provide a mechanism to use this metric as a useful indicator for refactoring weakly cohesive classes, thus showing its usefulness in improving class cohesion. Finally, we empirically validate LSCC. Using four open source software systems and eleven cohesion metrics, we investigate the relationship between LSCC, other cohesion metrics, and fault occurrences in classes. Our results show that LSCC is one of three metrics that explains more accurately the presence of faults in classes. LSCC is the only one among the three metrics to comply with important mathematical properties, and statistical analysis shows it captures a measurement dimension of its own. This suggests that LSCC is a better alternative, when taking into account both theoretical and empirical results, as a measure to guide the refactoring of classes. From a more general standpoint, the results suggest that class quality, as measured in terms of fault occurrences, can be more accurately explained by cohesion metrics that account for the degree of interaction between each pair of methods.

Keywords: object-oriented software quality, class cohesion, low-level design, method, attribute, refactoring, method-method interaction.

1. Introduction

The development of high-quality software is a primary goal in software engineering. Such software systems are likely to be stable and maintainable. During the development process, developers and managers typically apply quality metrics to assess and improve software quality. Cohesion, coupling, and complexity are common types of such metrics. The cohesion of a module indicates the extent to which the components of the module are related. A highly cohesive module performs a set of closely related actions and cannot be split into separate modules (Bieman and Ott 1994). Such a module is believed to be easier to understand, modify, and maintain than a less cohesive module (Briand et al. 2001a, Chen et al. 2002).

Classes are the basic units of design in object-oriented programs. Class cohesion refers to the relatedness of class members, that is, its attributes and methods. Several class cohesion metrics have been proposed in the literature. These metrics are based on information available during high- or low-level design phases. High-level design (HLD) cohesion metrics identify potential cohesion issues early in the HLD phase; see the HLD cohesion metrics proposed by Briand et al. (1999), Bansiya et al. (1999), Counsell et al. (2006), and Al Dallal and Briand (2010). These metrics do not support class refactoring activities that require information about the interactions between the methods and attributes in a class, because these interactions are not precisely known and defined in the HLD phase. Low-level design (LLD) cohesion metrics use finer-grained information than that used by the HLD cohesion metrics; see the LLD cohesion metrics proposed by Chidamber and Kemerer (1991), Chidamber and Kemerer (1994), Hitz and Montazeri (1995), Bieman and Kang (1995), Chen et al. (2002), Badri (2004), Wang (2005), Bonja and Kidanmariam (2006), Fernandez and Pena (2006), Chae et al. (2000), Chae et al. (2004), and Zhou et al. (2002). LLD cohesion information relies on precise knowledge of the relations between attributes and methods. Some LLD cohesion metrics (e.g., Henderson-Sellers 1996, Briand et al. 1998) are based on measuring attribute-method interactions by simply counting the total number of referenced attributes within the methods of a class, regardless of the allocation of these references. As a result, these metrics cannot *directly* identify weakly cohesive class members as required by refactoring. For example, in Figure 1, rectangles, circles, and links represent the methods, attributes, and use of attributes by methods, respectively, of a class including a weakly cohesive method m_3. Its removal would clearly improve class cohesion. Metrics based on counting the number of links (i.e., uses of attributes by methods) can indicate whether a class is strongly or weakly cohesive. However, these metrics fail to indicate the members of the class whose removal would improve class cohesion. Other LLD metrics (Chidamber and Kemerer 1991, Li and Henry 1993, Chidamber and Kemerer 1994, Bieman and Kang 1995, Hitz and Montazeri 1995, Badri 2004, Bonja and Kidanmariam 2006, Fernandez and Pena 2006) are based on measuring Method-Method Interactions (MMIs) by considering shared attributes between each pair of methods, which indicates to what extent each method is interconnected with other methods. This finer-grained information is important to help developers refactoring classes and detecting which methods to possibly remove (i.e., the methods that exhibit a few or even no interconnections with other methods). For example, when any of these MMI metrics is applied to measure the cohesion for the class represented in Figure 1, the interconnection between each pair of methods is calculated, and it is clearly found that method m_3 is weakly interconnected to the other methods in the class. This suggests that method m_3 should be removed from the class. Czibula and Serban (2006) and De Lucia et al. (2008) propose refactoring approaches based on MMI cohesion metrics.

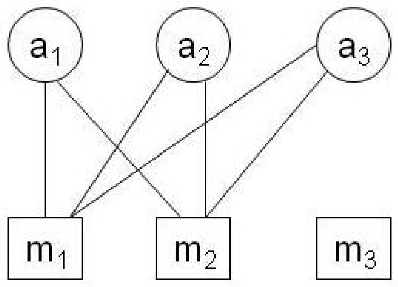

Figure 1: Sample representative graph for a hypothetical class

Several definitions for MMIs have been proposed. In some studies (Chidamber and Kemerer 1991, Chidamber and Kemerer 1994, Li and Henry 1993, Hitz and Montazeri 1995, Bieman and Kang 1995, Badri 2004), a pair of methods is considered to fully interact if they share at least one common attribute (i.e., a common attribute is referenced by each of the two methods). A counterintuitive case is when a pair of methods within the same class that share only one attribute is deemed to have the same cohesion degree as a pair of methods sharing all attributes. In other studies (Bonja and Kidanmariam 2006, Fernandez and Pena 2006), this problem is solved by using the number of shared attributes as a basis to measure the degree of interaction between each pair of methods. This is referred to as *similarity degree* and the authors provide formulas by which it can be calculated. However, these MMI similarity-based cohesion metrics have several limitations. The first limitation is that they are based on method-method similarity definitions, which are, in some cases, counterintuitive. For instance, in some studies (Bonja and Kidanmariam 2006, Fernandez and Pena 2006), a pair of methods, within the same class, that share relatively few attributes can be considered more cohesive than a pair of methods sharing more attributes. The second limitation is that MMI similarity-based cohesion metrics proposed to date have not been validated in terms of their mathematical properties and, in fact, violate key properties. The third limitation is that MMI similarity-based cohesion metrics proposed to date have not been empirically investigated in terms of their (1) usefulness as cohesion indicators, (2) correlations with other cohesion metrics, and (3) relationship with external software quality attributes (e.g., fault proneness, ease of change). Finally, the effect of considering (1) the interactions of methods through *transitive* method invocations, (2) class inheritance, and (3) different types of methods (i.e., setters, getters, constructor, and destructors) have neither been discussed nor empirically studied for MMI similarity-based cohesion metrics.

As a result, MMI metrics inherently consider the interaction between each pair of methods and this makes them particularly useful for refactoring when compared to other HLD and LLD metrics. However, MMI metrics proposed to date either do not consider the degree of interactions between methods, which weakens their ability to precisely indicate the degree of cohesion, or violate key cohesion properties. This highlights the need for an MMI metric that both considers the degree of interaction between each pair of methods and complies with key cohesion properties.

In this paper, we review and discuss some existing class cohesion metrics, with an emphasis on MMI metrics as they are better candidates to support refactoring, since they consider interactions between each pair of methods. We propose a LLD, Similarity-based Class Cohesion metric (LSCC), which is an MMI metric that accounts for the degree of interaction between each pair of methods. We demonstrate the use of LSCC as an indicator for refactoring weakly cohesive classes and derive computationally effective decision criteria for large scale analysis. The validity of a metric has to be studied both theoretically and empirically (Kitchenham et al. 1995). Theoretical validation tests whether the proposed metric complies with necessary properties of the measured attribute, though such properties are usually a matter of debate. Empirical validation tests whether what is being measured exhibits expected statistical relationships with other measures, such as measures of external quality attributes. Consistent with this general validation approach, LSCC is then validated from both theoretical and empirical standpoints.

Our theoretical validation involves analyzing the compliance of LSCC with the properties proposed by Briand et al. (1998), and our proofs show that LSCC does not violate any of them. The empirical validation involves eleven cohesion metrics, including the most common MMI cohesion metrics in the literature and LSCC, to classes selected from four open source Java systems for which fault report repositories are available. As a way to obtain indirect evidence that they measure what they purport to, we explore the correlations between the eleven considered metrics and the occurrence of faults in classes. The results show that LSCC is one of three metrics that better explains the presence of faults in classes, as compared to other proposed metrics. Additionally, it fares much better with that respect than other MMI cohesion metrics that do not consider the degree of interaction between each pair of methods. Taking into account both theoretical and empirical results, this indirectly suggests that LSCC is a better cohesion metric. This is based on the widely used and accepted assumption a (cohesion) metric that better explains the presence of faults is a better quality indicator and therefore a better constructed measure (e.g., Briand et al. 1998, Briand et al, 2001, Gyimothy et al. 2005, Aggarwal et al, 2007, and Marcus et al. 2008).

The major contributions of this paper are as follows:

1. Introducing an MMI-based cohesion metric (LSCC) that considers the degree of interaction between each pair of methods and, at the same time, satisfies key cohesion mathematical properties.
2. Proposing an objective, mathematical-based refactoring procedure based on the new cohesion metric.
3. Empirically validating LSCC by analyzing its correlation with eleven existing cohesion metrics and investigating its relationship, along with these other metrics, on the presence of faults in classes of four open source Java systems. The goal here was not to build fault prediction models but to study the relative fault prediction capability of LSCC and indirectly assess its strength as a quality indicator.
4. Theoretically and empirically studying the effect of including or excluding constructors, destructors, and access methods on LSCC calculation.

This paper is organized as follows. Section 2 reviews related work. Section 3 defines the proposed metric, and Section 4 shows its use as a class-refactoring indicator.

Section 5 discusses the theoretical validation of the proposed metric, and Section 6 illustrates several empirical case studies and reports and discusses results. Finally, Section 7 concludes the paper and discusses future work.

2. Related Work

In this section, we summarize a widely used set of mathematical properties that all class cohesion metrics are expected to satisfy and on which we will rely for our theoretical validation. In addition, we review and discuss several existing MMI class cohesion metrics for object-oriented systems and other related works in the area of software cohesion measurement.

2.1. Class cohesion metric necessary properties

Briand et al. (1998) defined four mathematical properties that provide a supportive underlying theory for class cohesion metrics. The first property, called *non-negativity and normalization*, states that a cohesion measure belongs to a specific interval [0, Max]. Normalization allows for easy comparison between the cohesion of different classes. The second property, called *null value and maximum value*, holds that the cohesion of a class equals 0 if the class has no cohesive interactions (i.e., interactions among attributes and methods of a class), while the cohesion of a class is equal to Max if all possible interactions within the class are present. The third property, called *monotonicity*, holds that the addition of cohesive interactions to the module cannot decrease its cohesion. The fourth property, called *cohesive modules*, holds that the merging of two unrelated modules into one module does not increase the module's cohesion. Therefore, given two classes, c1 and c2, the cohesion of the merged class c' must satisfy the following condition: cohesion(c')≤max {cohesion(c1), cohesion(c2)}. If a metric does not satisfy any of these properties, it is considered ill-defined (Briand et al. 1998). These properties were widely used to support the theoretical validation for several proposed class cohesion metrics (e.g., Briand et al. 1998, Zhou et al. 2002, Zhou et al. 2004, Al Dallal 2010).

2.2. MMI class cohesion metrics

Several metrics have been proposed in the literature to measure class cohesion during the HLD and LLD phases. These metrics use different underlying models and formulas. Table 1 provides definitions for ten of the most relevant works regarding MMI cohesion metrics. Other LLD and HLD metrics and less directly relevant work are briefly discussed in the subsequent subsection.

Class Cohesion Metric	Definition/Formula
Lack of Cohesion of Methods (LCOM1) (Chidamber and Kemerer 1991)	LCOM1= Number of pairs of methods that do not share attributes.
LCOM2 (Chidamber and Kemerer 1994)	*P*= Number of pairs of methods that do not share attributes. *Q*= Number of pairs of methods that share attributes. $LCOM2 = \begin{cases} P-Q & if\ P-Q \geq 0 \\ 0 & otherwise \end{cases}$
LCOM3 (Li and Henry 1993)	LCOM3= Number of connected components in the graph that represents each method as a node and the sharing of at least one attribute as an edge.
LCOM4 (Hitz and Montazeri 1995)	Similar to LCOM3 and additional edges are used to represent method invocations.
Tight Class Cohesion (TCC) (Bieman and Kang 1995)	TCC= Relative number of directly connected pairs of methods, where two methods are directly connected if they are directly connected to an attribute. A method *m* is directly connected to an attribute when the attribute appears within the method's body or within the body of a method invoked by method *m* directly or transitively.
Loose Class Cohesion (LCC) (Bieman and Kang 1995)	LCC=Relative number of directly or transitively connected pairs of methods, where two methods are transitively connected if they are directly or indirectly connected to an attribute. A method *m*, directly connected to an attribute *j*, is indirectly connected to an attribute *i* when there is a method directly or transitively connected to both attributes *i* and *j*.
Degree of Cohesion-Direct (DC_D) (Badri 2004)	DC_D= Relative number of directly connected pairs of methods, where two methods are directly connected if they satisfy the condition mentioned above for TCC or if the two methods directly or transitively invoke the same method.
Degree of Cohesion-Indirect (DC_I) (Badri 2004)	DC_I= Relative number of directly or transitively connected pairs of methods, where two methods are transitively connected if they satisfy the same condition mentioned above for LCC or if the two methods directly or transitively invoke the same method.
Class Cohesion (CC) (Bonja and Kidanmariam 2006)	CC= Ratio of the summation of the similarities between all pairs of methods to the total number of pairs of methods. The similarity between methods *i* and *j* is defined as: $Similarity(i,j) = \frac{\lvert I_i \cap I_j \rvert}{\lvert I_i \cup I_j \rvert}$, where I_i and I_j are the sets of attributes referenced by methods *i* and *j*, respectively.
Class Cohesion Metric (SCOM) (Fernandez and Pena 2006)	SCOM= Ratio of the summation of the similarities between all pairs of methods to the total number of pairs of methods. The similarity between methods *i* and *j* is defined as: $Similarity\ (i,j) = \frac{\lvert I_i \cap I_j \rvert}{\min(\lvert I_i \rvert, \lvert I_j \rvert)} \cdot \frac{\lvert I_i \cup I_j \rvert}{l}$, where *l* is the number of attributes

Table 1: Definitions of the considered MMI class cohesion metrics

Al Dallal (2010a) proved that none of the lack of cohesion metrics (i.e., LCOM1, LCOM2, LCOM3, and LCOM4) satisfies the normalization property. In addition, LCOM3 and LCOM4 do not satisfy the *null value and maximum value* property. The metrics LCOM1, LCOM2, LCOM3, LCOM4, TCC, LCC, DC_D, and DC_I do not distinguish between pairs of methods that share different numbers of attributes. Thus, counter-intuitively, a pair of methods within the same class that shares only one attribute has the same cohesion degree as a pair of methods sharing all attributes. The CC metric measures the cohesion more precisely than the aforementioned metrics, because it considers the number of shared attributes between each pair of methods. However, CC does not satisfy the monotonicity property in some cases. That is, when a cohesive interaction is added to a class, the counter-intuitive result may be a class with a lower CC value, as depicted in classes A and B, given in Figure 2, where rectangles, circles, and links represent methods, attributes, and the use of the attributes by the methods, respectively. This occurs because the addition of a cohesive interaction may increase the similarity between a pair of methods and decrease the similarity between other pairs of methods. In this case, the cohesion increases if the summation of the similarities between pairs of methods increases, and vice versa. Similar to CC, SCOM measures the cohesion more precisely than the aforementioned metrics because it considers the number of shared attributes between each pair of methods. However, SCOM does not satisfy the monotonicity property in some cases, as depicted in classes C and D, given in Figure 2. Both CC and SCOM neither consider transitive method-method interactions nor account for inheritance or different types of methods, and they have not been empirically validated against external quality attributes such as fault occurrences.

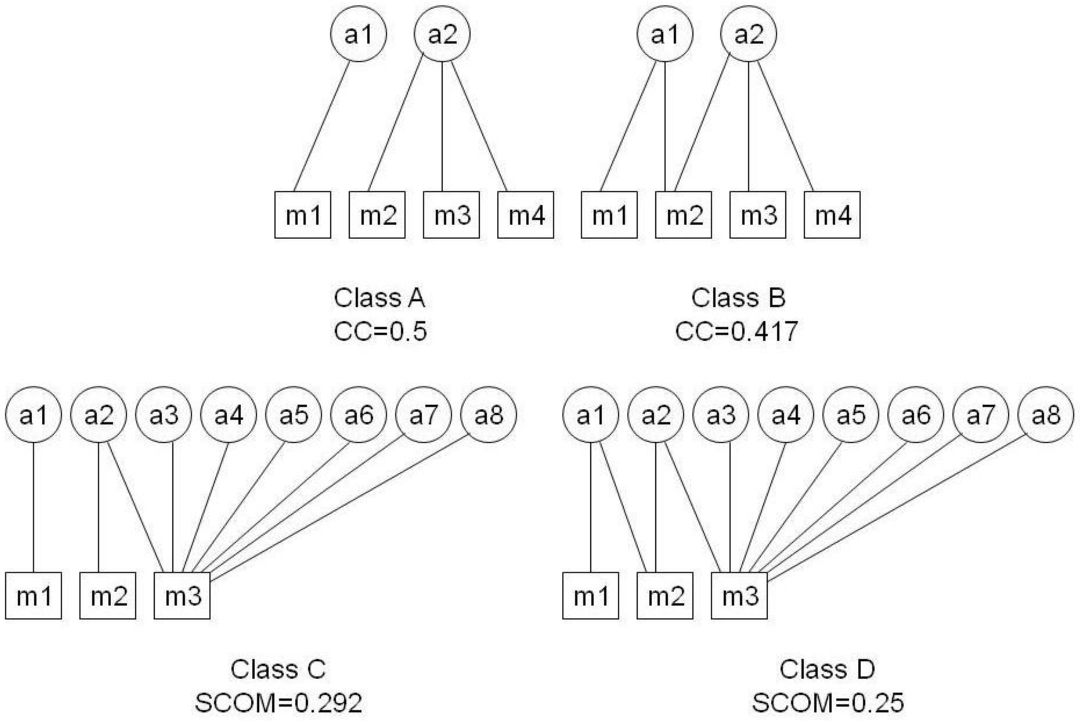

Figure 2: Classes with different method-method connectivity patterns

2.3. Overview of other relevant work

Yourdon et al. (1979) proposed seven levels of cohesion. These levels include coincidental, logical, temporal, procedural, communicational, sequential, and functional. The cohesion levels are listed in ascending order of their desirability. Since then, several cohesion metrics have been proposed for procedural and object-oriented programming languages. Different models have been used to measure the cohesion of procedural programs, such as the control flow graph (Emerson 1984), the variable dependence graph (Lakhotia 1993), and program data slices (Ott and Thuss 1993, Bieman and Ott 1994, Al Dallal 2007, Al Dallal 2009). Cohesion has also been indirectly measured by examining the quality of structure designs (Troy and Zweben 1984, Bieman and Kang 1998).

Several LLD class cohesion metrics have been proposed in the literature, including the ones discussed in the previous subsection. Henderson-Sellers (Henderson-Sellers 1996) proposed a metric called LCOM5 that measures a lack of cohesion in methods by considering the number of methods referencing each attribute. In this case, LCOM5=(*a*-*kl*)/(*l*-*kl*), where *l* is the number of attributes, *k* is the number of methods, and *a* is the number of distinct attributes accessed in the methods of a class. Briand et al. (1998) proposed a cohesion metric called Coh that computes cohesion as the ratio of the number of distinct attributes accessed in the methods of a class. Wang et al. (2005) introduced a Dependence Matrix-based Cohesion (DMC) class cohesion metric based on a dependency matrix that represents the degree of dependence among the instance variables and methods in a class. Chen et al. (2002) used dependence analysis to explore attribute-attribute, attribute-method, and method-method interactions. They measured cohesion as the relative number of interactions. Chae et al. (2000) propose a metric called Cohesion Based on Member Connectivity (CBMC) that does not consider only the number of interactions, but also the patterns of the interactions between the methods in a class. The metric considers the ratio of the number of glue methods to the number of methods of interest. The number of glue methods equals to the minimum number of methods required such that their removal causes the method-attribute interaction graph to become disjoint. Zhou et al. (2002) introduce ICBMC, an improved version of CBMC, that considers the cut sets instead of glue methods. The cut set is the minimum set of edges such that their removal causes the method-attribute interaction graph to become disjoint. Chae et al. (2004) consider the effects of dependent attributes on the measurement of cohesion. Dependent attributes are the attributes whose values are obtained using other attributes. Zhou et al. (2004) and Etzkorn et al. (2003) analyze and compare several LLD class cohesion metrics.

Several HLD class cohesion metrics have also been proposed in the literature, including Cohesion Among Methods in a Class (CAMC), Normalized Hamming Distance (NHD), Scaled NHD (SNHD), and Similarity-based Class Cohesion (SCC). The CAMC metric (Bansiya et al. 1999) uses a parameter-occurrence matrix that has a row for each method and a column for each data type used at least once as the type of a parameter in at least one method in the class. The value in row *i* and column *j* in this matrix equals 1 when the *i*th method has a parameter of the *j*th data type, and it equals 0 otherwise. The CAMC metric is defined as the ratio of the total number of 1s in the matrix to the total size of the matrix. The NHD metric (Counsell et al. 2006) uses the same parameter-occurrence matrix used by CAMC. The metric calculates the average parameter agreement between each pair of methods. The parameter agreement between a pair of methods is defined as the number of entries in which the corresponding rows in the parameter-occurrence matrix match. SNHD (Counsell et al. 2006) is a metric that represents the closeness of the NHD metric to the maximum value of NHD, as compared to the minimum value. The SCC metric (Al Dallal and Briand 2010) uses a matrix called a Direct Attribute Type (DAT) matrix, which has a row for each method and a column for each distinct parameter type that matches an attribute type. The value in row *i* and column *j* in the matrix equals 1 when the *j*th data type is a type of at least one of the parameters or return of the *i*th method, and it equals 0 otherwise. The SCC metric is the weighted average of four different metrics that consider method-method, attribute-attribute, and attribute-method direct and transitive interactions. Method-Method through Attributes Cohesion (MMAC) considers method-method interactions, and it is defined as the ratio of the summation of the similarities between all pairs of methods to the total number of possible pairs of methods. The similarity between a pair of methods is defined as the ratio of shared parameter types that match attribute types to the number of parameter types represented in the DAT matrix.

In summary, several class cohesion metrics have been introduced in the literature. Some of these metrics are based on measuring MMIs defined during the LLD phase.

However, these metrics have one or more of the following limitations: (1) they do not consider the number of shared attributes; (2) they do not satisfy key mathematical cohesion properties; (3) they do not consider class inheritance, different types of methods, or transitive interactions; and (4) they have not been empirically validated in terms of their relationships to external quality attributes such as fault occurrences. To address these issues, in this paper, we propose a metric called LLD Similarity-based Class Cohesion (LSCC). LSCC is based on a precise MMI definition that satisfies widely accepted class cohesion properties and is useful as an indicator for restructuring weakly cohesive classes. Also, LSCC can easily account for class inheritance and direct and transitive interactions. In addition, LSCC differentiate between different types of methods (i.e., access methods, constructors, and destructors).

3. LLD Similarity-based Class Cohesion (LSCC)

The LLD Similarity-based Class Cohesion (LSCC) proposed in this paper considers MMIs modeled in a matrix, denoted here as Method-Attribute Reference (MAR). The matrix and the corresponding metric are defined as follows.

3.1. Model definition

The Method-Attribute Reference (MAR) matrix is a binary $k \times l$ matrix, where k is the number of methods and l is the number of attributes in the class of interest. The MAR matrix has rows indexed by the methods and columns indexed by the attributes, and so for $1 \leq i \leq k$, $1 \leq j \leq l$,

$$m_{ij} = \begin{cases} 1 & \text{if } i\text{th method references } j\text{th attribute,} \\ 0 & \text{otherwise} \end{cases}$$

The information required to construct this matrix is obtained by analyzing the source code of the class of interest. Note that a reference to a variable which name matches the name of both a local variable and an attribute is not considered a reference to the attribute unless it is explicitly indicated (e.g., using the *this* keyword in Java). The matrix implicitly models method-method interactions. A cohesive method-method interaction is represented in the MAR matrix by two rows that share binary values of 1 in a column.

The MAR matrix can account for transitive interactions caused by method invocations. In this case, the binary value 1 in the matrix indicates that the attribute is directly or transitively referenced by the method. An attribute is transitively referenced by method m_j if the attribute is directly referenced by method m_i and if method m_j directly or transitively invokes method m_j. For example, Table 2 shows the MAR matrix corresponding to the sample *PrintValues* Java class shown in Figure 3. The MAR matrix shows that methods *PrintX* and *PrintY* directly reference attributes *x* and *y*, respectively, and methods *printXZ* and *printYZ* directly reference attribute *z*. In addition, the table shows five binary values of 1, highlighted in boldface for clarification purposes, to represent the transitive references of attribute *x* by method *printXZ*, attribute *y* by method *printYZ*, and attributes *x*, *y*, and *z* by method *printXZY*.

```
public class PrintValues{
        private int x,y,z;
        public void printX(int a){
                if (a==0) System.out.println("x is not initialized");
                else System.out.println(x=a);
        }
        public void printY(int a){
                if (a==0) System.out.println("y is not initialized");
                else System.out.println(y=a);
        }
        public void printXZ(int a,int b){
                printX(a);
                if (b==0) System.out.println("z is not initialized");
                else System.out.println(z=a);
        }
        public void printYZ(int a,int b){
                printY(a);
                if (b==0) System.out.println("z is not initialized");
                else System.out.println(z=a);
        }
        public void printXZY(int a,int b,int c){
                printXZ(a,b);
                printY(c);
        }
}
```
Figure 3: Sample *PrintValues* Java class

	x	y	z
printX	1	0	0
printY	0	1	0
printXZ	**1**	0	1
printYZ	0	**1**	1
printXZY	**1**	**1**	**1**

Table 2: The MAR matrix for the *PrintValues* class including transitive interactions

3.2. The metric

The similarity between two items is the collection of their shared properties. In the context of the MAR matrix introduced in Section 3.1, the similarity between two rows quantifies the cohesion between a pair of methods. As in the study of Al Dallal and Briand's (2010), the similarity between a pair of rows is defined as the number of entries in a row that have the same binary values as the corresponding elements in the other row. The normalized similarity, denoted as *ns(i,j)*, between a pair of rows *i* and *j* is defined as the ratio of similarity between the two rows to the number of entities Y in the row of the matrix, and it is formally defined as follows:

$$ns(i,j) = \frac{\sum_{x=1}^{Y}(m_{ix} \wedge m_{jx})}{Y} \quad , \qquad (1)$$

where $\wedge$ is the logical *and* relation.

Cohesion refers to the degree of similarity between module components. LSCC is the average cohesion of all pairs of methods. Using the MAR matrix, the LSCC of a class C consisting of *k* methods and *l* attributes is formally defined as follows:

$$LSCC(C)=\begin{cases}0 & \text{if } l=0 \text{ and } k>1,\\ 1 & \text{if } (l>0 \text{ and } k=0) \text{ or } k=1,\\ \dfrac{2}{k(k-1)}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k} ns(i,j) & \text{otherwise.}\end{cases} \tag{2}$$

LSCC is undefined for the meaningless case of a class with no attributes and methods. If a class with several methods does not have any attributes, the methods will be considered unrelated and the cohesion will be the minimum value. If a class that has attributes does not have methods, the attributes declare the structure of a class that does not have behavior. Our interpretation for this case is that we would expect all the attributes to describe the features of the same object. Therefore, the class is expected to be fully cohesive. The general definition for a cohesive module is that it performs a single task and cannot be easily split (Bieman and Ott 1994). Assuming that each method performs a cohesive task, a class that has a single method is associated with a single task, namely the task performed by its method, and cannot be easily split. Therefore, the cohesion value for such a class must be the maximum. By substituting Formula 1 into Formula 2, the LSCC of class C is calculated in the case of a class with multiple methods as follows:

$$LSCC(C)=\frac{2}{lk(k-1)}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\sum_{w=1}^{l}(m_{iw}\wedge m_{jw}) \tag{3}$$

The following metric is an alternative form of the LSCC metric, which facilitates the analysis of the metric and speeds up its computation:

Proposition 3.1. For any class *C*,

$$LSCC(C)=\begin{cases}0 & \text{if } l=0 \text{ and } k>1,\\ 1 & \text{if } (l>0 \text{ and } k=0) \text{ or } k=1,\\ \dfrac{\sum_{i=1}^{l} x_i(x_i-1)}{lk(k-1)} & \text{otherwise.}\end{cases} \tag{4}$$

Note that x_i is the number of 1s in the *i*th column of the MAR matrix.

Proof: By definition, when l=0 or $k\leq1$, Equations 3 and 4 are equal. Otherwise, for the *i*th column, there are $x_i(x_i$-1)/2 similarities between the methods (i.e., cases where two methods access the same attribute), and therefore:

$$LSCC(C)=\frac{2}{lk(k-1)}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\sum_{w=1}^{l}(m_{iw}\wedge m_{jw})=\frac{2}{lk(k-1)}\frac{\sum_{i=1}^{l}x_i(x_i-1)}{2},$$

which equals the above formula. ■

For example, using Formula 4 and the MAR matrix given in Table 2, leaving out transitive interactions, the LSCC metric for the *PrintValues* class is calculated as follows:

$$LSCC(PrintValues)=\frac{1(0)+1(0)+2(1)}{3(5)(4)}=0.033$$

In this case, the value of LSCC appears to be low because ignoring the transitive interactions leaves only two methods, namely *printXZ* and *printYZ*, to share a common attribute and all other methods to be disjoint. When transitive interactions are considered, the LSCC metric for the *PrintValues* class is calculated as follows:

$$LSCC(PrintValues) = \frac{3(2)+3(2)+3(2)}{3(5)(4)} = 0.3$$

Typically, object oriented classes include constructors and access methods (i.e., setter and getter methods). In some object-oriented programming languages, such as C++, classes can have destructors too. Usually, the constructor method provides initial values for most or all of the class attributes. Therefore, typically, the similarity, defined in formula (1), between the constructor and any other method *m* is equal to or higher than the similarity between method *m* and any other method in the class. As a result, including constructors can cause LSCC to increase artificially. The same argument applies for destructors, though destructors are less problematic, because they typically do not reference most if not all attributes. An access method references by definition one attribute, and therefore, the similarity, defined by LSCC, between the access method and each non-access method is relatively low. In addition, the similarity between each pair of setter methods equals zero. The same applies for each pair of getter methods. Therefore, the presence of access methods causes LSCC to decrease artificially. As a result, when applying LSCC, constructors, destructors, and access methods should be excluded based on theoretical grounds.

Note that the same argument is not applicable for CC and SCOM. In CC, the similarity between a pair of methods is defined as the ratio of the number of shared attributes by the two methods to the number of distinct attributes referenced by either of the two methods. As a result, unlike the similarity defined for LSCC, the similarity, defined for CC, between a method *m* and a constructor, a destructor, or an access method can be less than, greater than, or equal to the similarity between method *m* and any other method in the class. Therefore, the average similarities between method *m* and each of the other methods including the constructor can be less than, greater than, or equal to that when the constructor is excluded. In conclusion, ignoring the constructor, destructor, or access methods from the computation of CC causes unpredictable effect on CC values. Similar to CC, the similarity defined by SCOM, between two methods also depends on the number of shared attributes and the number of distinct attributes referenced by the two methods. As a result, contrary of LSCC, the impact of ignoring the access methods, constructors, and destructors from the computation of CC and SCOM is hard to theoretically determine.

To account for class inheritance, all directly and transitively accessible inherited methods and attributes must be included in the MAR matrix. The inherited methods can be extracted with source code analysis, though dynamic binding introduces complications as it is not considered in the MAR matrix since the source code analysis is performed statically. Including inherited attributes and methods allows for measuring the cohesion of the class as a whole, whereas excluding them results in measuring the cohesion of the class locally. This means that including or excluding the inherited attributes and methods depends on the measurement purpose. Inherited methods do not reference the attributes of the inheriting class. Therefore, including only the inherited methods decreases the average similarities between each pair of methods, and consequently, decreases the LSCC value. On the other hand, inherited attributes are supposed to be referenced by some of methods in the inheriting class. Therefore, the change in the LSCC value after including only the inherited attributes depends on whether the added references increase or decrease the average similarities between each pair of methods. Finally, the change in the value of LSCC, when including both the inherited attributes and methods, depends on (1) the change in the similarities between each pair of inheriting methods, (2) the average similarities between each pair of inherited methods, and (3) the average similarities between each pair of inheriting and inherited methods. The first and third factors can cause the value of LSCC to increase or decrease. On the other hand, the second factor causes the value of LSCC to decrease because, in this case, the number of considered attributes is increased, whereas the number of references and the number of methods remain the same.

One advantage of LSCC over LCOM1, LCOM2, TCC, LCC, $DC_{D,}$ and DC_I is that it considers the number of shared attributes between each pair of methods in the class. In addition, it is better than CC and SCOM in the sense that it (1) accounts for transitive interactions and class inheritance, (2) differentiates between different types of methods, and (3) satisfies the mathematical cohesion properties, as will be discussed later in Section 5.

4. Refactoring Threshold

Refactoring aims at enhancing the code understandability and maintainability, and it refers to the process of changing an existing object-oriented software code to improve its internal structure while preserving its external behavior (Fowler 1999). Tokuda and Batory (2001) identify several refactoring benefits including automating design changes, reducing testing efforts, simplifying designs, assisting validation, and experimenting new designs. Fowler (1999) describes when and how to perform several refactoring scenarios. Researchers use design metrics, including cohesion, to either validate their refactoring techniques (e.g., Keeffe and Cinneide 2006) or guide refactoring activities (e.g., De Lucia et al. 2008 and Czibula and Serban 2006). Demeyer and Verelst (2004) identify *Move Method* and *Extract Class* among the refactoring activities that can improve the cohesion and coupling of an existing code. *Move Method* refers to moving a method from one class to the class in which the method is used most, and *Extract Class* refers to creating a new class to include related attributes and methods currently existing in the original class (Fowler 1999). De Lucia et al. (2008) propose a cohesion-based refactoring technique that considers the structural and semantic similarity between each pair of methods. The structural similarity is based on the similarity metric originally proposed by Bonja and Kidanmariam (2006), whereas the semantic similarity is based on the conceptual cohesion metric (Marcus and Poshyvanyk 2005) that considers the comments and identifiers included in the source code. De Lucia et al. (2008) suggest an experimental-based value for the refactoring threshold. However, the external validity of this value is necessary limited due to the specificities of the classes selected to run the experiment. In this section, we propose a theory-based refactoring threshold using LSCC. Consequently, the proposed threshold is independent of any experimental context. Two refactoring scenarios are considered. In the first scenario, the threshold based on LSCC is introduced to support *Move Method* (i.e., decide whether a certain method should be removed from a class, and probably, moved to another class, to improve overall software cohesion). In the second scenario, the threshold is used to guide *Extract Class* (i.e., decide whether a class should be split into several classes to enhance overall software cohesion). Though we realize that there are other factors to consider as well besides cohesion, using multiple criteria for class feature assignment is out of the scope of this paper.

4.1. Move Method Refactoring

A method should be removed from a class if its interaction with the other methods is weak to the extent that the method's inclusion in the class weakens its overall cohesion. In this case, the cohesion of the class after method removal is higher than that before the method removal. Let us denote the method that we would like to consider removing as *m*, the class before the removal of the method as C_{old}, the class after the removal of the method as C_{new}, the number of attributes and methods in C_{old} as *l* and *k*, respectively, the number of 1s in the *i*th column (*i*th attribute) in the MAR matrix representing class C_{old} as x_i, and the value of the entry in row *m* and column *i* in the MAR matrix (i.e., the reference of attribute *i* by method *m*) as y_i. As discussed in the proof of Formula 4, the numerator of the formula equals twice the number of similarities between the methods of the class under consideration, and it therefore equals the summation of twice the number of similarities between the methods of the class excluding method *m* and twice the number of similarities between method *m* and the other methods of the class, which is denoted as *α*. In other words, *α* is the degree of interaction between method *m* and the other methods in class C_{old}, and it is calculated as $\alpha = 2\sum_{i=1}^{l} x_i y_i$. The method should be removed when:

$$LSCC(C_{old}) < LSCC(C_{new}) \Rightarrow \frac{\sum_{i=1, i\neq m}^{l} x_i(x_i - 1) + \alpha}{lk(k-1)} < \frac{\sum_{i=1, i\neq m}^{l} x_i(x_i - 1)}{l(k-1)(k-2)} \Rightarrow \alpha < \frac{2\sum_{i=1, i\neq m}^{l} x_i(x_i - 1)}{k-2}$$

Instead of computing LSCC for the class before and after the method removal and comparing the results, the above inequality eases the required computations, a clear benefit when facing many methods and a large number of classes. The inequality is applicable when there are more than two methods in the class. When the class includes only two methods, developers are advised to move a method to a more appropriate class if the degree of interaction between the considered method and the methods of the target class is greater than the degree of interaction between the considered method and the other method in the considered class. Intuitively, it is rather obvious that developers should not be advised to move a method from a class when this class includes one method. LSCC is consistent with this expectation as, given the definition in Formula 4, such a class has the maximum possible value.

For example, for the class given in Figure 3, the above inequality can be applied to decide whether to move method *printX*. In this case, we refer to the MAR matrix given in Table 2, x_1y_1=1*2=2, where x_1 is the cell value in the first column of the matrix in the row representing the methods *printX*, and y_1 is the number of 1s in the first column of the matrix in the rows representing the other methods. Similarly, x_iy_i can be calculated for each of the three columns in the matrix. As a result, the value for *α* for method *printX* equals $2\sum_{i=1}^{l} x_i y_i$ = 2[1(2)+0(3)+0(3)] = 4. In this case, the value for the other side of the inequality equals 2[2(1)+3(2)+3(2)]/(5-2) = 9.33. Since the inequality is true, we conclude that the method should be removed, and in this case, the cohesion of the class will increase. Similarly, the values for *α* for each of the other methods given in Table 2 equal 4, 8, 8, and 12, respectively. The corresponding values for the other side of the inequality equal 9.33, 6.67, 6.67, and 4, respectively. Since the inequality is true only for *printY*, as well as *printX*, as shown above, we conclude that these two methods should be removed from the class, and the other methods should be retained. This result matches our expectation, since removal of these two methods from the class increases its LSCC value from 0.3 to 0.55.

Ignoring the constructors, destructors, and access methods, when calculating LSCC, prevents mistakenly recommending these methods to be removed from the class. That is, if the access methods are included in the LSCC calculation, the refactoring threshold can mistakenly indicate that the access methods have to be removed from the class because the similarity between each of such methods and each of the other methods is expected to be low. In refactoring, cohesion has to be considered locally (i.e., inherited methods and attributes should be ignored). Otherwise, including such methods and attributes can cause recommending their removal from the class, which would be absurd. Finally, considering transitive interactions decreases the chances of mistakenly recommending the removal of methods invoked by other methods in the class.

Note that the proposed refactoring threshold *α* is only applicable when LSCC for the class before moving the method is greater than zero. If none of the pairs of methods in the class before moving the method shares a common attribute, the class LSCC value will be zero. In this case, during refactoring, developers are advised to move the methods of the class to more appropriate classes.

4.2. Extract Class Refactoring

Based on the definition of the LSCC metric, a threshold can be defined and used as a support for deciding whether a class should be kept as is or divided into several classes. A class could be partitioned into several classes if the cohesion of these classes is greater than the cohesion of the original class. Each of these classes consists of a set of methods moved from the original class, and therefore, the Move Method refactoring, tackled in Section 4.1, is a special case of the more general Extract Class refactoring. The cohesion of software consisting of several classes is defined as the weighted cohesion of the classes, where the numerator part of the weight of a class equals *l* if *k* equals one; otherwise, the numerator part of the weight of a class is equal to the denominator of Formula 4, given in Section 3.2, for that class. For all classes, the denominator part of the weight is equal to the summation of the denominator part of Formula 4 for all classes. For simplicity, the case in which we must decide whether a class should be split into two classes is considered here. The same concept can be applied to a decision about partitioning a class into more than two classes. For a class *C*, given that the numerator and denominator of Formula 4 are denoted by *N*(*C*) and *D*(*C*), respectively, when class C is split into two classes C_1 and C_2, the weighted cohesion of the two classes (i.e., the resulting cohesion of the cluster consisting of the two classes) equals:

$$\frac{LSCC(C_1)\times D(C_1)+LSCC(C_2)\times D(C_2)}{D(C_1)+D(C_2)}=\frac{N(C_1)+N(C_2)}{D(C_1)+D(C_2)}$$

As discussed in the proof of Formula 4, *N*(*C*) is twice the number of similarities between the methods of class *C*, and it therefore equals the summation of (1) twice the number of similarities between the methods of class C_1, (2) twice the number of similarities between the methods of class C_2, and (3) twice the number of similarities between the methods of class C_1 and the methods of class C_2, which is denoted as α. As a result, the numerator of Formula 4 for class C before splitting equals $N(C_1)+N(C_2)+\alpha$. In this case, class C should be split into two classes C_1 and C_2 when:

$$\frac{N(C_1)+N(C_2)}{D(C_1)+D(C_2)} > \frac{N(C_1)+N(C_2)+\alpha}{D(C)}. \tag{5}$$

The term α defined above equals $2\sum_{i=1}^{l} x_i y_i$, where l is the number of attributes in class C, and x_i and y_i are the number of 1s in column i of the MAR matrix in the rows representing the methods of classes C_1 and C_2, respectively. The term α indicates the degree of interaction between the methods of the two classes. Recall that the formula used to compute α is the same for both *Extract Class* refactoring addressed here and *Move Method* refactoring addressed in Section 4.1. The only difference is in the value of x_i, which equals the number of 1s in column i in the rows representing the moved out methods. That is, in *Move Method* refactoring, only one row corresponding to the moved method is considered, whereas several rows corresponding to the set of moved methods are considered in *Extract Class* refactoring. By substituting the numerator and denominator of Formula 4 in Inequality 5, the threshold for α is defined as follows:

$$\alpha < \frac{[D(C)-D(C_1)-D(C_2)][N(C_1)+N(C_2)]}{D(C_1)+D(C_2)} = \frac{[l.k(k-1)-l_1.k_1(k_1-1)-l_2.k_2(k_2-1)][\sum_{i=1}^{l_1} x_i(x_i-1)+\sum_{i=1}^{l_2} y_i(y_i-1)]}{l_1.k_1(k_1-1)+l_2.k_2(k_2-1)},$$

where l, l_1, l_2, k, k_1, and k_2 are the numbers of attributes and methods in classes C, C_1, and C_2, respectively. Because the weight of the cohesion of a class equals l if k equals one, the above inequality is defined for all possible cases as follows:

$$\alpha < \begin{cases} \dfrac{[l.k(k-1)-l_1-l_2][l_1+l_2]}{l_1+l_2} = l.k(k-1)-l_1-l_2 & \text{if } \mathrm{k}_1 = 1 \text{ and } \mathrm{k}_2 = 1 \\ \dfrac{[l.k(k-1)-l_1-l_2.k_2(k_2-1)][l_1+\sum_{i=1}^{l_2} y_i(y_i-1)]}{l_1+l_2.k_2(k_2-1)} & \text{if } \mathrm{k}_1 = 1 \text{ and } \mathrm{k}_2 > 1 \\ \dfrac{[l.k(k-1)-l_1.k_1(k_1-1)-l_2][\sum_{i=1}^{l_1} x_i(x_i-1)+l_2]}{l_1.s_1(s_1-1)+l_2} & \text{if } \mathrm{k}_1 > 1 \text{ and } \mathrm{k}_2 = 1 \\ \dfrac{[l.k(k-1)-l_1.k_1(k_1-1)-l_2.k_2(k_2-1)][\sum_{i=1}^{l_1} x_i(x_i-1)+\sum_{i=1}^{l_2} y_i(y_i-1)]}{l_1.k_1(k_1-1)+l_2.k_2(k_2-1)} & \text{if } \mathrm{k}_1 > 1 \text{ and } \mathrm{k}_2 > 1 \end{cases} \tag{6}$$

Inequality (6) reduces the complexity of the computations required for class refactoring decision. That is, instead of (1) computing LSCC for each of the considered classes, (2) computing the weighted average of the calculated LSCC values, (3) computing LSCC for the resulting class, and (4) comparing the results of Steps 2 and 3 to decide whether to split the class, inequality 6 can be used directly to make the refactoring decision. For example, for the class given in Figure 3, the above inequality (6) can be applied to decide whether to split the class in two, with one containing the methods *printX* and *printY* and the other containing the other three methods. In this case, we refer to the MAR matrix given in Table 2, x_1y_1=1*2=2, where x_1 is the number of 1s in the first column of the matrix in the rows representing the methods *printX* and *printY*, and y_1 is the number of 1s in the first column of the matrix in the rows representing the other methods. Similarly, x_iy_i can be calculated for each of the three columns in the matrix. As a result, $\alpha= 2\sum_{i=1}^{3} x_i y_i$ =2[1(2)+1(2)+0(3)]=8. In addition, referring to the data given in Table 2, note that l=3, l_1=2 (i.e., numbers of attributes referenced by the class containing the methods *printX* and *printY*), l_2=3, k=5, k_1=2, and k_2=3, and therefore, the inequality in (6) can be computed as follows:

$$8 < \frac{[3(5)(5-1)-2(2)(2-1)-3(3)(3-1)][(0+0)+(2(2-1)+2(2-1)+3(3-1))]}{2(2)(2-1)+3(3)(3-1)}$$

Since the inequality is true, we conclude that the class should be split as specified above. In contrast, the following application of the inequality shows that the class should not be split into two classes, namely, one containing the methods *printX*, *printY*, and *printZ* and the other containing the other two methods. In this case, $\alpha= 2\sum_{i=1}^{3} x_i y_i$ =2[2(1)+2(1)+1(2)]=12. In addition, referring to the data given in Table 2, note that l=3, l_1=3, l_2=3, k=5, k_1=3, and k_2=2, and therefore, the inequality is as follows: $12 > \frac{[3(5)(5-1)-3(3)(3-1)-3(2)(2-1)][(2+0+0)+(0+2+2)]}{3(3)(3-1)+3(2)(2-1)}$, which indicates that the class should not be split as specified above.

Note that inequality (6) is only applicable when the LSCC value of the class before being split is greater than zero. If none of the pairs of methods in the class before being split shares a common attribute, the LSCC value of the class will be zero. In this case, the class must be strongly considered as a candidate for refactoring.

4.3. Refactoring Case Study

Extract Class refactoring, in which several methods are moved out from a class, is a generalized case of the Move Method refactoring, in which a single method is moved out. In this case study, we assessed the proposed Move Method refactoring criterion based on LSCC in order to determine whether it leads to adequate decisions. The same results follow for the Extract Class refactoring criterion that considers moving more than a method, because both criteria are based on the same principle.

To assess whether our proposed LSCC-based, Move Method refactoring criterion (Section 4.1) is really appropriate to decide about refactoring, we applied it to classes randomly selected from JHotDraw version 7.4.1 (JHotDraw 2010), an open source framework developed as an exercise for using design patterns. JHotDraw is well-known for its good design (Czibula and Serban 2006) and consists of 576 classes. To a large extent, with high confidence, JHotDraw can therefore be considered a well-designed system. In such a system, methods are expected to be encapsulated in their most appropriate classes. As a result, a method that would artificially be moved from its original class to an arbitrary target class is highly likely to be weakly related to the target class members. Such a case should therefore be detected by a well-defined refactoring criterion—such as the one we defined—when applied on the modified target class. In order to assess our refactoring criterion we test this conjecture on JHotDraw classes to determine whether it is suggested the newly moved methods are misplaced and should be taken out.

A research assistant, who previously was not involved in this research, randomly selected 130 JHotDraw classes. Among them, 65 classes were randomly selected as target classes to which methods would be moved. The restrictions placed on the choice of these target classes were that they (1) are not utility classes (i.e., not classes defined to provide common reusable methods) and (2) include several attributes and several methods other than the access methods and constructors. In other words, each of the selected target classes constitutes an abstract data type encapsulating methods and attributes. As a result, each time a class was sampled randomly from the pool of the 130 classes and its compliance with the two criteria was assessed. The classes that fitted both criteria were included in the set of target classes, and the other classes were discarded. This process terminated when the required 65 classes were identified. The discarded and remaining classes were used as source classes from which classes were moved.

A method was randomly selected from each of the classes not previously selected as target classes. The restrictions placed on the choice of these methods were that they are not constructors or access methods and their size was not trivial based on an arbitrary threshold of 20 LOC. Each of the selected methods was moved using the Move Method refactoring process defined by Fowler (1999) to a different class selected randomly from the 65 target classes. Note that moving a method normally requires adjusting the method to make it work in the target class (e.g., if the moved method references attributes of the source class, new get methods may be needed in the source class interface). As a result, each of the 130 selected classes was used once as either a source or a target class. Our Move Method refactoring criterion was then applied on each of the artificially moved methods in the target 65 classes to test whether the resulting decision correctly matched the expectation that the misplaced method should be removed from its class.

Using the LSCC-based inequality in Section 4.1, our results showed that our Move Method refactoring criterion was able to detect that all the methods artificially moved to target classes should be moved out. This empirically supports our claims, previously made based on theoretical arguments, that LSCC is an appropriate cohesion metric to guide refactoring, which was stated earlier as our main motivation to define it. Since they are based on similar principles, similar results would be expected for the Extract Class refactoring criterion.

4.4. Applications

This section does not introduce a refactoring approach. Instead, it proposes a threshold that can be used in guiding class refactoring activities. This threshold is based on theoretical analysis of LSCC, and therefore, it eliminates the limitation of the approach proposed by De Lucia et al. (2008), where an experimental threshold value is proposed. There are several applications for our theory-based threshold. A developer can apply the introduced inequalities to point out hot spots that suggest refactoring opportunities. For example, in a real setting, both sides of the inequality proposed in Section 4.1 can be calculated for each method in each class in the system. The classes can be ranked according to the percentage of methods for which the inequality holds. Classes with higher percentages of such methods are considered hot refactoring spots and have to be given higher priority during the refactoring process.

The refactoring approach proposed by De Lucia et al. (2008) considers both structural and semantic cohesion. However, as discussed earlier in this section, that approach is based on experimental threshold values. It can however be extended by applying the same ideas proposed in this section for calculating the threshold for both structural and semantic cohesion metrics. The combination of both thresholds can be used to decide whether a set of methods should be extracted to form another class or whether a specific method should be removed from a class. The extended approach has to be evaluated experimentally to compare its results with the refactoring results obtained using the experimental-based thresholds. Such a study is left open for further research.

4.5. Limitations

The proposed threshold is based on structural information of the source code, whereas semantic information is ignored. In some cases, this can cause refactoring out methods that are structurally weakly cohesive but strongly cohesive semantically. The second application introduced in the previous subsection can be applied to overcome this limitation.

Refactoring classes requires not only accounting for cohesion, but also accounting for other quality attributes, such as coupling (e.g., Demeyer and Verelst 2004). Coupling refers to the degree to which classes are connected to each other. Software developers aim at increasing class cohesion and decreasing coupling between classes. Cohesion and coupling are the two quality factors identified to be improved by refactoring activities (Demeyer and Verelst 2004). When performing refactoring tasks, developers have to keep the balancing between the improvement of both cohesion and coupling because improving one of these quality attributes can weaken the other. Despite the importance of considering these two factors together, in this paper, we focus on cohesion, and we leave the consideration of coupling and other factors open for future research.

Finally, effective class refactoring also requires accounting for the attributes (i.e., whether to keep the attributes in a class or to remove them). This paper considers only MMIs, which indirectly considers the relations between attributes. The refactoring threshold specified above can be improved by accounting for the degree of interactions between attributes as well as methods. In summary, a comprehensive refactoring approach should account for coupling between classes and structural and semantic cohesion among the methods and attributes of a class. The approach should aim at restructuring the class to maximize its internal cohesion and minimize its coupling with other classes. Thresholds have to be computed to decide whether to apply *Extract Class* or *Move Method*.

To obtain the best solution for class splitting, α must be calculated for each possible splitting scenario. Given a class consisting of k methods, there are $C(k,n) = k!/[k!(k-n)!]$ possible ways to split the class into two classes, when one of them includes n methods and the other includes the remaining methods. Considering possible values of n, there are $\sum_{n=1}^{\lfloor \frac{k}{2} \rfloor} C(k,n)$ possible ways to split the class into two classes, a usually large number which requires efficient computations. This justifies the importance of introducing Inequality (6) to speed up the computations required for refactoring decisions. Furthermore, the number of possible ways to partition a class increases much more when considering scenarios with more than two classes, which means that calculating α for each of the possible cases is not feasible for classes consisting of a large number of methods. In the future, we plan to introduce a clustering-based algorithm that uses α to partition the methods into several groups such that each group represents a class and overall software cohesion is improved. This approach can be compared with the one proposed by De Lucia et al. (2008).

5. Theoretical Validation

We validate LSCC using the properties for a class cohesion metric proposed by Briand et al. (1998) and discussed in Section 2.1.

Property LSCC.1: The LSCC metric complies with the non-negativity and normalization property.

Proof: The minimum value for the LSCC metric for a class is 0 when the class has either (1) several methods and no attributes or (2) several methods such that none of their pairs share a common attribute. The maximum value for the LSCC metric for a class is 1 when the class has (1) one or more attributes and no methods, (2) one method, or (3) several methods and one or more attributes with each pair of methods sharing all attributes in the class (i.e., each method references all attributes in the class). As a result, the LSCC metric ranges over the interval [0, 1], and it therefore complies with the non-negativity and normalization property.■

Property LSCC.2: The LSCC metric complies with the null and maximum values property.

Proof: Given a class with a set of methods and attributes, if none of the pairs of methods share a common attribute (that is, the class has no MMIs), the value of the LSCC metric will be 0. Alternatively, if each attribute is shared between each pair of methods (that is, the class features all possible MMIs), the value of the LSCC metric will be 1 (that is, the maximum possible value). Hence, the LSCC metric complies with the null and maximum values property.■

Property LSCC.3: The LSCC metric complies with the monotonicity property.

Proof: The addition of an MMI to the MAR matrix is represented by changing an entry value from 0 to 1 in a column that has at least one entry value of 1. Changing such an entry value from 0 to 1 increases the number of 1s in the column, which increases the numerator value in Formula 4. An increase in the numerator in Formula 4 increases the value of the LSCC metric, because the denominator does not change unless the size of the matrix changes. As a result, the addition of a cohesive interaction represented in the MAR matrix always increases the LSCC value, which means that the LSCC metric complies with the monotonicity property.

Property LSCC.4: The LSCC metric complies with the cohesive module property.

Proof: The merging of two unrelated classes *c1* and *c2* implies that none of the methods in each of the two classes are shared and that none of them share common attributes. Therefore, the number of rows and columns in the MAR matrix of the merged class equals the sum of the number of rows and columns in the MAR matrices of the unrelated classes. The number of 1s in each column in the MAR matrix of the merged class equals the number of 1s in the corresponding column in the MAR matrices of the unrelated classes. Therefore, for the MAR $k \times l$ matrix representing class *c1*, the MAR $m \times n$ matrix representing class *c2*, and the MAR $(k + m) \times (l + n)$ matrix representing the merged class *c3*:

$$\sum_{i=1}^{l} x_i(x_i - 1) + \sum_{i=1}^{n} x_i(x_i - 1) = \sum_{i=1}^{l+n} x_i(x_i - 1)$$

Suppose that LSCC(*c1*)≥LSCC(*c2*), then:

$$\frac{\sum_{i=1}^{l} x_i(x_i - 1)}{lk(k-1)} \ge \frac{\sum_{i=1}^{n} x_i(x_i - 1)}{mn(m-1)} \Rightarrow mn(m-1)\sum_{i=1}^{l} x_i(x_i - 1) \ge lk(k-1)\sum_{i=1}^{n} x_i(x_i - 1)$$

$$\Rightarrow [mn(m-1) + lk(k-1)]\sum_{i=1}^{l} x_i(x_i - 1) \ge lk(k-1)[\sum_{i=1}^{n} x_i(x_i - 1) + \sum_{i=1}^{l} x_i(x_i - 1)]$$

$$\Rightarrow \frac{\sum_{i=1}^{l} x_i(x_i - 1)}{lk(k-1)} \ge \frac{\sum_{i=1}^{n} x_i(x_i - 1) + \sum_{i=1}^{l} x_i(x_i - 1)}{mn(m-1) + lk(k-1)} > \frac{\sum_{i=1}^{n} x_i(x_i - 1) + \sum_{i=1}^{l} x_i(x_i - 1)}{mn(m-1) + lk(k-1) + (l+n)km + (lm+nk)(k+m-1)}$$

$$\Rightarrow \frac{\sum_{i=1}^{l} x_i(x_i - 1)}{lk(k-1)} > \frac{\sum_{i=1}^{l+n} x_i(x_i - 1)}{(l+n)(m+k)(k+m-1)} \Rightarrow LSCC(c1) > LSCC(c3)$$

So, *Max*{LSCC(*c1*),LSCC(*c2*)}>LSCC(*c3*).
This means that the LSCC metric complies with the cohesive module property. ■

By showing that a cohesion metric fulfills expected mathematical properties, the chances that the metric is a meaningful class cohesion indicator increase and it is therefore more likely to relate to external quality indicators such as fault occurrences in classes. This, however, is not necessarily guaranteed and must be empirically investigated. The following section reports on a large-scale empirical investigation that explores whether LSCC explains more of the statistical variation in fault occurrences than other MMI cohesion metrics. If confirmed, this would indirectly provide additional evidence that LSCC is a better defined metric.

6. Empirical Validation

When assessing whether LSCC is a useful contribution, several criteria must be considered together: its mathematical properties, its accuracy as an indicator of external quality attributes (e.g., faults), and its relationships with existing metrics. The mathematical properties are considered in Section 5, and in this section, the other two criteria are addressed. We present three analyses. The first explores the correlations among eleven cohesion metrics, including LSCC and well known other metrics, and applies principal component analysis (Dunteman 1989) to explore the orthogonal dimensions within this set of cohesion metrics. The goal is to confirm that LSCC is indeed contributing new information. The second analysis explores the extent to which the eleven class cohesion metrics, including LSCC, can explain the presence of faults in classes. The goal of this analysis is to determine whether there is empirical evidence, either direct or indirect, that LSCC is indeed a well-defined measure and as such is better than existing, comparable cohesion metrics. In this study, we rely on the widely accepted and used assumption, used in many studies (e.g., Briand et al. 1998, Briand et al, 2001, Gyimothy et al. 2005, Aggarwal et al. 2007, Olague et al. 2007, and Marcus et al. 2008), that the metric that predicts faulty classes more precisely is expected to be a better quality indicator. Note that building a fault prediction model is *not* one of the empirical study goals because this goal would require considering other factors such as size, coupling, and complexity, which is out of the scope of this paper. The third analysis compares the fault prediction results when accounting for transitive

MMIs with previous results. The goal of this analysis is to determine whether considering transitive MMIs in LSCC improves its ability to predict faulty classes and would therefore suggest this leads to better measurement.

6.1. Software systems and metrics

We chose four Java open source software systems from different domains: Art of Illusion version 2.5 (Illusion 2009), GanttProject version 2.0.5 (GanttProject 2009), JabRef version 2.3 beta 2 (JabRef 2009), and Openbravo version 0.0.24 (Openbravo 2009). Art of Illusion consists of 488 classes and about 88 K lines of code (LOC), and it is a 3D modeling, rendering, and animation studio system. GanttProject consists of 496 classes and about 39 KLOC, and it is a project scheduling application featuring resource management, calendaring, and importing or exporting (MS Project, HTML, PDF, spreadsheets). JabRef consists of 599 classes and about 48 KLOC, and it is a graphical application for managing bibliographical databases. Openbravo consists of 452 classes and about 36 KLOC, and it is a point-of-sale application designed for touch screens. We chose these four open source systems randomly from http://sourceforge.net. The restrictions placed on the choice of these systems were that they (1) are implemented using Java, (2) are relatively large in terms of the number of classes, (3) are from different domains, and (4) have available source code and fault repositories. The same systems were considered by Al Dallal and Briand (2010) for validating the SCC HLD-based metric.

We selected ten MMI cohesion metrics to compare with LSCC. The metrics are CC, SCOM, LCOM1, LCOM2, LCOM3, LCOM4, TCC, LCC, DC_D, and DC_I. The former two metrics consider measuring the degree of MMIs, whereas the rest do not. They were selected because they have already been extensively studied and compared to each other (Briand et al. 1999, Briand et al. 2000, Briand et al. 2001b, Marcus et al. 2008, Al Dallal and Briand 2010, Al Dallal 2010b). Therefore, our results can be compared to those obtained in previous empirical studies.

We applied the considered metrics for 1355 selected classes among 2035 classes from the four open source systems. We excluded all classes for which at least one of the metrics is undefined. For example, classes consisting of single methods were excluded because their LCOM1, LCOM2, TCC, LCC, DC_D, DC_I, CC, and SCOM values are undefined. In addition, classes not consisting of any attributes were excluded because their TCC, LCC, DC_D, DC_I, CC, and SCOM values are undefined. An advantage of the LSCC metric is that it is defined in all cases, as discussed in Section 3.2. Therefore, none of the classes were excluded because of an undefined LSCC value. Exclusion of the classes that have undefined cohesion values using the metrics under consideration allows us to perform the same analysis for all metrics on the same set of classes and therefore compare their results in an unbiased manner. Interfaces were also excluded because LLD metrics, including LSCC, are undefined in this case.

We developed our own Java tool to automate the cohesion measurement process for Java classes using eleven metrics, including LSCC. The tool analyzed the Java source code, extracted the information required to build the matrices, and calculated the cohesion values using the eleven considered metrics. Table 3 shows descriptive statistics for each cohesion measure including the minimum, 25% quartile, mean, median, 75% quartile, maximum value, and standard deviation. As indicated in

Briand et al. (2001), LCOM1, LCOM2, LCOM3, and LCOM4 are not normalized, and they feature extreme outliers due to accessor methods that typically reference single attributes. The 25% quartile, mean, median, and 75% quartile for LSCC are less than those for CC and SCOM due to the corresponding normalized similarity definitions. LSCC is normalized by dividing the similarity degree by the total number of attributes of the class, whereas CC and SCOM are normalized by dividing the similarity degree by a smaller factor that depends on the number of attributes referenced by at least one of the considered pair of methods. Table 3 also shows that the 25% quartile, mean, median, and 75% quartile for LSCC, CC, and SCOM are less than those for the other MMI metrics. This occurs because the degree of similarity between a pair of methods sharing at least one common attribute, as calculated using LSCC, CC, and SCOM, is greater than zero and less than or equal to 1, whereas it always equals 1 when using LCOM1, LCOM2, LCOM3, LCOM4, TCC, LCC, DC_D, and DC_I.

Although we showed how to include inherited methods and attributes when measuring LSCC, the following analyses do not consider inheritance. As discussed in Section 4, considering inheritance misleads refactoring decisions, which is the main focus of this paper. In addition, LSCC is based on static analysis and dynamic analysis to accurately measure cohesion (as it was done by Briand et al. in 2004 for coupling) in the presence of inheritance would be an entirely new problem to be addressed by future work. Last, our only goal here is to compare LSCC with other cohesion metrics, which do not account for inheritance, in terms of correlations and fault predictions. Using inheritance in the definition of LSCC would therefore make such comparisons difficult.

Metric	Min	25%	Mean	Med	75%	Max	Std Dev
LSCC	0	0.03	0.44	0.22	1.00	1.00	0.44
CC	0	0.08	0.31	0.19	0.42	1.00	0.31
SCOM	0	0.09	0.36	0.25	0.58	1.00	0.34
LCOM1	0	2.00	57.88	9.00	36.50	3401.00	200.57
LCOM2	0	0.00	38.98	2.00	22.00	2886.00	159.25
LCOM3	0	1.00	1.25	1.00	1.00	17.00	1.01
LCOM4	0	1.00	1.18	1.00	1.00	17.00	0.90
TCC	0	0.24	0.52	0.50	0.80	1.00	0.34
LCC	0	0.33	0.64	0.71	1.00	1.00	0.36
DC_D	0	0.26	0.53	0.50	0.80	1.00	0.33
DC_I	0	0.33	0.65	0.71	1.00	1.00	0.36

Table 3: Descriptive statistics for the cohesion measures

6.2. Correlation and principal component analyses

Principal Component Analysis (PCA) (Dunteman 1989) is a technique used here to identify and understand the underlying orthogonal dimensions that explain the relations between the cohesion metrics (Marcus et al. 2008). For each pair of cohesion metrics under consideration, we used the Mahalanobis Distance (Barnett and Lewis 1994) to detect outliers, and we found that the removal of outliers does not lead to significant differences in the final PCA results. We calculated the nonparametric

Spearman correlation coefficient (Siegel and Castellan 1988) among the considered cohesion metrics. Table 4 shows the resulting correlations among the considered metrics accounting for all four systems. They are all statistically significant (p-value < 0.0001). Coefficients showing strong correlations (≥0.8) are highlighted in boldface. The results show that the correlations among CC, SCOM, TCC, and DC_D are strong. Among these metrics, the correlations between CC and SCOM, among TCC, LCC, DC_D, and DC_I, between LCOM1 and LCOM2, and between LCOM3 and LCOM4 are greater than 0.9. This is because each of these groups of metrics have similar definitions. The other cohesion metrics, including LSCC, are weakly or moderately inter-correlated.

Metric	CC	SCOM	LCOM1	LCOM2	LCOM3	LCOM4	TCC	LCC	DC_D	DC_I
LSCC	0.54	0.59	-0.60	-0.58	-0.22	-0.17	0.49	0.42	0.47	0.42
CC	1.00	**0.91**	-0.59	-0.75	-0.14	-0.09	**0.85**	0.72	**0.84**	0.71
SCOM		1.00	-0.67	**-0.80**	-0.24	-0.17	**0.87**	0.77	**0.85**	0.76
LCOM1			1.00	**0.80**	0.33	0.27	-0.56	-0.42	-0.55	-0.41
LCOM2				1.00	0.35	0.28	-0.73	-0.61	-0.72	-0.60
LCOM3					1.00	**0.90**	-0.15	-0.11	-0.15	-0.11
LCOM4						1.00	-0.14	-0.13	-0.14	-0.13
TCC							1.00	**0.90**	**0.99**	**0.90**
LCC								1.00	**0.89**	**0.99**
DC_D									1.00	**0.90**

Table 4: Spearman rank correlations among the cohesion metrics

To obtain the principal components (PCs), we used the *varimax* rotation technique (Jolliffe 1986, Snedecor and Cochran 1989) in which the eigenvectors and eigenvalues (loadings) are calculated and used to form the PC loading matrix. Table 5 shows the PCA results: the loading matrix shows four PCs that capture 93.76% of the data set variance. In addition, it shows the eigenvalues (i.e., measures of the variances of the PCs), their percentages, and the cumulative percentage. High coefficients (loadings) for each PC indicate which are the influential metrics contributing to the captured dimension. Coefficients above 0.5 are highlighted in boldface in Table 5. Based on an analysis of these coefficients, the resulting PCs can then be interpreted as follows:

PC1: LSCC, CC, SCOM, TCC, LCC, DC_D, and DC_I. These MMI metrics measure cohesion directly and they are normalized.
PC2 and PC3: LCOM1, LCOM2, LCOM3, and LCOM4. These MMI metrics measure a lack of cohesion. In addition, they are not normalized and can have extreme outliers.
PC4: LSCC: This is our new metric, which measures the degree of interactions between each pair of methods and is the only MMI considered metric that ignores constructors, destructors, and access methods.

The PCA results show that LSCC metric captures a measurement dimension of its own as it is the only significant factor in PC4, though it also contributes to PC1. This supports the fact that LSCC captures cohesion aspects that are not addressed by any of the cohesion metrics considered in this analysis, thus confirming the results of correlation analysis.

	PC1	PC2	PC3	PC4
Eigenvalue	5.61	2.35	1.44	0.92
Percent	50.99	21.36	13.06	8.34
Cum. Per.	50.99	72.35	85.41	93.76
LSCC	**0.56**	-0.08	0.27	**0.67**
CC	**0.85**	0.12	0.15	0.29
SCOM	**0.91**	0.08	0.12	0.24
LCOM1	-0.30	**0.76**	**-0.52**	0.20
LCOM2	-0.32	**0.77**	-0.48	0.25
LCOM3	-0.35	**0.69**	**0.60**	-0.12
LCOM4	-0.33	**0.67**	**0.64**	-0.09
TCC	**0.96**	0.17	<0.01	-0.13
LCC	**0.87**	0.27	-0.16	-0.29
DC_D	**0.95**	0.18	-0.01	-0.13
DC_I	**0.87**	0.28	-0.16	-0.30

Table 5: The loading matrix

6.3. Predicting faults in classes

To study the relationship between the values of the collected metrics and the extent to which a class is prone to faults, we applied logistic regression (Hosmer and Lemeshow 2000), a standard and mature statistical method based on maximum likelihood estimation. This method is widely applied to predict fault-prone classes (Briand et al. 1998, Briand et al, 2001, Gyimothy et al. 2005, and Marcus et al. 2008), and though other analysis methods, such as the methods discussed by Briand and Wuest (2002), Subramanyam and Krishnan (2003), and Arisholm et al. (2009), could have been used, this is out of the scope of this paper. In logistic regression, explanatory or independent variables are used to explain and predict dependent variables. A dependent variable can only take discrete values and is binary in the context where we predict fault-prone classes. The logistic regression model is univariate if it features only one explanatory variable and multivariate when including several explanatory variables. In this case study, the dependent variable indicates the presence of one or more faults in a class, and the explanatory variables are the cohesion metrics. Univariate regression is applied to study the fault prediction capability of each metric separately, whereas multivariate regression is applied to study the combined fault prediction of several cohesion metrics; the goal is to determine whether LSCC improves the fit of these combinations. Univariate regression analysis was applied to compare the fault prediction power of (1) the eleven metrics under consideration, (2) each of the metrics that consider the degree of interactions between methods (i.e., LSCC, CC, and SCOM) with and without considering transitive MMIs, and (3) each of the metrics that consider the degree of interactions between methods including and excluding constructors and access methods. The goal of the first study is to compare the fault prediction power of LSCC with those of the other MMI metrics. The second study explores whether it is better, from a fault prediction perspective, to consider transitive MMI interactions in LSCC, CC, and SCOM. Finally, the third study investigates whether it is better, from a fault prediction perspective, to consider constructors and access methods in the class cohesion measurement using LSCC, CC, and SCOM. Note that CC and SCOM were originally defined to include access methods and constructors, but here we nevertheless investigate whether excluding these methods improves the performance of these metrics in predicting faulty classes. When excluding access methods and constructors, the remaining number of methods in some classes can be less than two. LSCC is defined for such classes, whereas CC and SCOM are undefined. It is found that 116 (8.6%) of the considered classes consist of less than two methods when access methods are excluded. In addition, 415 (30.6%) of the considered classes include less than two methods when both access methods and constructors are excluded. This relatively high percentage of classes for which CC and SCOM are undefined demonstrates one of their drawbacks. The applicability of CC and SCOM, as well as other cohesion metrics, is thoroughly studied by Al Dallal (2010c). Al Dallal suggests values for classes whose cohesion was previously undefined using CC, SCOM, and other cohesion metrics, and shows that these values increase the applicability of the metrics and also improve the precision of most metrics in indicating class quality. We used these values to make CC and SCOM applicable for all considered classes when excluding access methods and constructors. This allows us to perform regression analysis on the same set of considered classes.

We collected fault data for the classes in the considered software systems from publicly available fault repositories. The developers of the considered systems used an on-line Version Control System (VCS) to keep track of source code changes. The changes, called revisions, are due to either detected faults or required feature improvements. Each revision is associated with a report including the revision description and a list of classes involved in this change. Two research assistants, one with a B.Sc. in computer science and six years of experience in software development activities and the other with a B.Sc. and Master both in computer science; each of them, manually and independently traced the description of each revision and identified faults. The first author of this paper compared the manual results and rechecked the results in which the two assistants differed to choose the correct one. Finally, we used the lists of classes involved in changes due to detected faults to count the number of faults in which each class is involved. For each of the considered systems, Table 6 reports the number considered classes, number of faulty classes, and the sum of the number of faults involved in the classes. Figure 4 shows the distribution of the number of faults reported for the classes of the four systems. The fault repositories include reports about the detected and fixed faults and specify which classes are involved in these faults. We manually traced the reports and counted the number of faults detected in each class. We classified each class as being fault-free or as having at least one fault, as a small percentage of classes contain two faults or more. Ideally, class cohesion should be measured before each fault occurrence and correction, and used to predict this particular fault occurrence. However, not only this would mean measuring cohesion for dozens of versions (between each fault correction) for each system, but we would not be able to study the statistical relationships of a set of faults with a set of consistent cohesion measurements for many classes. Our cohesion measurement is based on the latest version of the source code, after fault corrections, and is therefore an approximation. This is however quite common in similar research endeavors (e.g., Briand et al. 1998, Briand et al, 2001, Gyimothy et al. 2005, and Marcus et al. 2008) and is necessary to enable statistical analysis.

	Art of Illusion	Gantt-Project	JabRef	Open-bravo	All systems
No. of considered classes	399	337	300	319	1355
No. of faulty classes	189	188	233	249	859
No. of faults	213	460	494	357	1524
No. of classes having one fault	176	119	183	189	667

Table 6: Fault data extracted from the VCS of the considered systems

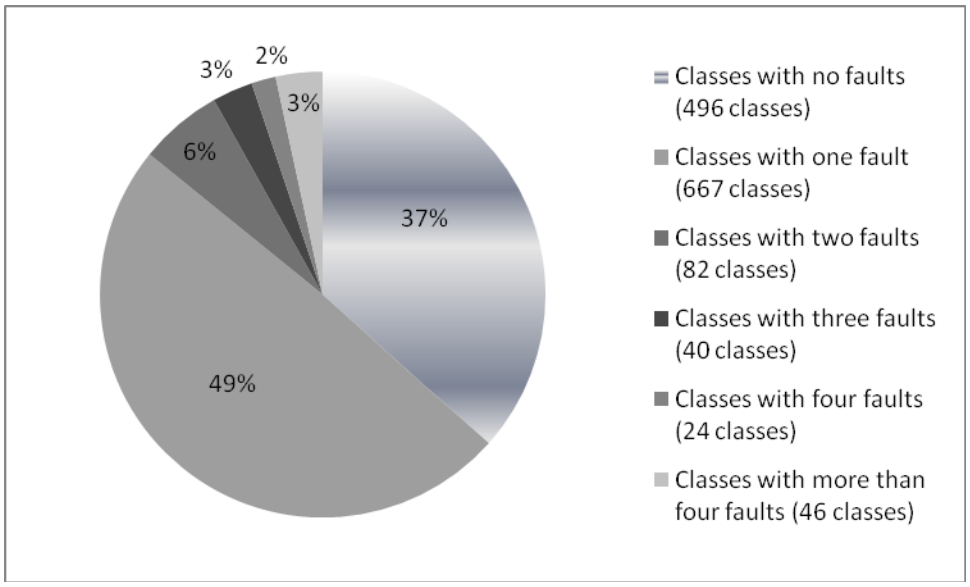

Figure 4: The distribution of the number of faults detected in classes

The results of the univariate regression studies are reported in Table 7. The results for LSCC, CC, and SCOM shown in the first nine rows of Table 7 only consider direct MMIs, whereas the results for these metrics reported in the last three rows of Table 7 consider direct and transitive MMIs. The results reported in the first three rows are for the following cases (1) $LSCC_1$ accounts for constructors and access methods, (2) $LSCC_2$ ignores access methods, and (3) $LSCC_3$ ignores both constructors and access methods. Note that the third case is the one recommended in Section 3.2, and its results are highlighted in boldface in Table 7. Similarly, CC_1, CC_2, CC_3, $SCOM_1$, $SCOM_2$, and $SCOM_3$ are defined and their results are reported in the next six rows. The results for LSCC, CC, and SCOM, when transitive MMIs are considered, are reported in the last three rows of Table 7. In the computation of each of the three metrics, we included the transitive MMIs in the computation of the metric version that had the best results among the ones reported in the first nine rows of Table 7. For example, transitive MMI are considered in the computation of $LSCC_3$, because the fault prediction results of $LSCC_3$ are the best among the ones reported in the first three rows. Similarly, transitive MMIs are considered in the computations of CC_1 and $SCOM_1$ because the results for these versions of the metrics are the best among the three studied versions. Estimated regression coefficients are reported, as well as their 95% confidence intervals. The larger the absolute value of the coefficient is, the stronger the impact (positive or negative, according to the sign of the coefficient) of the metric on the probability of a fault being detected in a class. The considered metrics have significantly different standard deviations as shown in Table 3. Therefore, to help compare the coefficients, we standardized the explanatory variables by subtracting the mean and dividing by the standard deviation and, as a result, they all have an equal variance of 1 and the coefficients reported in Table 7 are also standardized. These coefficients represent the variation in standard deviations in the dependent variable when there is a change of one standard deviation in their corresponding independent variable. The p-value is the probability of the coefficient being different from zero by chance, and is also an indicator of the accuracy of the coefficient estimate: The larger the p-value, the larger the confidence interval for the coefficient. A common practice is to use *odd ratios* (Hosmer and Lemeshow 2000) to help interpret coefficients as those are not linearly related to the probability of fault occurrences. In our context, an odd ratio captures how less (more) likely it is for a fault to occur when the corresponding (lack of) cohesion metric augments by one standard deviation. We report odd ratios and their 95% confidence interval in Table 7. As an example, for LSCC, the probability of fault occurrence when there is an increase of one standard deviation in LSCC is estimated to decrease by 37%. Those can be easily compared across cohesion metrics. We use a typical significance threshold (α=0.05) to determine whether a metric is a statistically significant fault predictor. To avoid the typical problem of inflation of type-I error rates in the context of multiple tests, we used a corrected significance threshold using the Bonferroni adjustment procedure: α/11=0.0045 (Abdi 2007).

Type	Metric	Std. Coeff.	Odd ratio	Std. Error	95% Confidence Interval Coeff.	95% Confidence Interval odd ratio	p-value	Precision	Recall	ROC area
Considering degree of interaction	LSCC$_1$	-0.46	0.63	0.06	[-0.57,-0.35]	[0.57,0.71]	< 0.0001	63.1	65.5	64.3
	LSCC$_2$	-0.45	0.64	0.06	[-0.56,-0.34]	[0.57,0.71]	< 0.0001	63.7	65.8	60.5
	LSCC$_3$	**-0.57**	**0.64**	**0.06**	**[-0.56,-0.33]**	**[0.57,0.72]**	**< 0.0001**	**65.3**	**65.3**	**64.5**
	CC$_1$	-0.45	0.64	0.06	[-0.57,-0.34]	[0.57,0.71]	< 0.0001	63.0	65.5	63.6
	CC$_2$	-0.43	0.65	0.06	[-0.55,-0.32]	[0.58,0.73]	< 0.0001	63.7	65.8	60.1
	CC$_3$	-0.53	0.59	0.06	[-0.64,-0.41]	[0.53,0.66]	< 0.0001	63.3	63.9	62.7
	SCOM$_1$	-0.49	0.61	0.06	[-0.61,-0.38]	[0.54,0.68]	< 0.0001	63.4	65.8	63.6
	SCOM$_2$	-0.39	0.68	0.06	[-0.50,-0.28]	[0.61,0.76]	< 0.0001	40.8	63.8	59.2
	SCOM$_3$	-0.53	0.59	0.06	[-0.65,-0.42]	[0.52,0.66]	< 0.0001	60.5	63.3	63.0
Not considering degree of interaction	LCOM1	1.07	2.93	0.2	[0.68,1.47]	[1.97,4.36]	< 0.0001	40.8	63.8	62.8
	LCOM2	1.23	3.41	0.24	[0.75,1.7]	[2.12,5.49]	< 0.0001	40.8	63.8	62.0
	LCOM3	0.06	1.07	0.06	[-0.06,0.18]	[0.95,1.2]	0.29	40.8	63.8	49.6
	LCOM4	0.03	1.03	0.06	[-0.09,0.14]	[0.92,1.15]	0.65	40.8	63.8	48.5
	TCC	-0.4	0.67	0.06	[-0.51,-0.29]	[0.6,0.75]	< 0.0001	56.6	62.9	60.7
	LCC	-0.31	0.73	0.06	[-0.43,-0.2]	[0.65,0.82]	< 0.0001	40.8	63.8	58.8
	DC$_D$	-0.38	0.69	0.06	[-0.49,-0.26]	[0.61,0.77]	< 0.0001	40.8	63.8	60.0
	DC$_I$	-0.29	0.75	0.06	[-0.41,-0.18]	[0.66,0.84]	< 0.0001	40.8	63.8	58.4
Considering transitive MMIs	LSCC	-0.56	0.57	0.06	[-0.67,-0.44]	[0.51,0.64]	< 0.0001	65.4	65.0	63.3
	CC	-0.44	0.64	0.06	[-0.56,-0.33]	[0.57,0.72]	< 0.0001	63.2	65.6	62.2
	SCOM	-0.48	0.62	0.06	[-0.59,-0.36]	[0.55,0.7]	< 0.0001	63.2	65.5	62.7

Table 7: Univariate logistic regression results for the eleven metrics under consideration

To evaluate the prediction accuracy of logistic regression models, we used the traditional precision and recall evaluation criteria (Olson and Delen 2008). Precision is defined as the number of classes correctly classified as faulty divided by the total number of classes classified as faulty. It measures the percentage of faulty classes correctly classified as faulty. Recall is defined as the number of classes correctly classified as faulty divided by the actual number of faulty classes. It measures the percentage of faulty classes correctly or incorrectly classified as faulty. Such criteria, however, require the selection of a probability threshold to predict classes as faulty. Following the recommendation in Briand et al. (2000), a class is classified as faulty if its predicted probability of containing a fault is higher than a threshold that is selected such that the percentage of classes that are classified as faulty is roughly the same as the percentage of classes that are actually faulty.

To evaluate the performance of a prediction model regardless of any particular threshold, we used the receiver operating characteristic (ROC) curve (Hanley and McNeil 1982). In a fault prediction context, the ROC curve is a graphical plot of the ratio of classes correctly classified as faulty versus the ratio of classes incorrectly classified as faulty at different thresholds. The area under the ROC curve shows the ability of the model to correctly rank classes as faulty or non-faulty. A 100% ROC area represents a perfect model that correctly classifies all classes. The larger the ROC area, the better the model in terms of classifying classes. The ROC curve is often considered a better evaluation criterion than standard precision and recall, as selecting a threshold is always somewhat subjective.

To obtain a more realistic assessment of the predictive capacities of the metrics, we used cross-validation, a procedure in which the data set is partitioned into *k* sub-samples. The regression model is then built and evaluated *k* times. Each time, a different sub-sample is used to evaluate the precision, recall, and ROC area of the model, and the remaining sub-samples are then used as training data to build the regression model.

The results in Table 7 lead to the following observations:

1. Except for LCOM3 and LCOM4, all of the cohesion metrics are statistically significant (i.e., their coefficients are significantly different from 0), even when accounting for Bonferroni's adjustment ($\alpha = 0.05/11 = 0.0045$).
2. LSCC, recommended in Section 3.2, is the best metric among the eleven MMI metrics under consideration in terms of ROC area.
3. The logistic regression results for the metrics that consider the degree of interaction between each pair of methods are very close to each other. However, based on the fact that CC and SCOM violate key cohesion properties, that may lead to incorrect cohesion assessment, LSCC is therefore a preferable cohesion metric. Note that, in Section 6.2, it is shown that LSCC captures a distinct cohesion dimension.
4. LSCC is considerably better in terms of ROC area than metrics that do not account for the degree of interaction between each pair of methods. From this perspective, LSCC is a better cohesion metric, when compared to LCOM1, LCOM2, LCOM3, LCOM4, TCC, LCC, DC_D, and DC_I, because it explains better the presence of faulty classes.

5. LSCC has the largest standardized coefficient among MMI cohesion metrics. This is confirmed by a relatively smaller odd ratio (0.64), thus suggesting that an increase in LSCC has the strongest impact on reducing fault occurrence probability among MMI metrics. To compare the odd ratios of inverse cohesion metrics—which have coefficients above one—with LSCC, one must divide one by these odd ratios to obtain a comparable value (i.e., the odd ratio when there is a decrease of one standard deviation in lack of cohesion). For example, with LCOM2 which has the largest effect among inverse metrics, this odd ratio is 1/1.23 = 0.81.
6. As expected, the estimated regression coefficients are positive for the inverse cohesion measures LCOM1, LCOM2, LCOM3, and LCOM4, whereas they are negative for straight cohesion measures LSCC, CC, SCOM, TCC, LCC, DC_D, and DC_I.
7. Among MMI metrics under consideration, LCOM1, LCOM2, LCOM3, LCOM4, LCC, DC_D, and DC_I are the worst metrics in terms of precision, and LCOM4 is the worst metric in terms of ROC area.
8. Ignoring constructors and access methods in the computation of LSCC considerably increases its standard coefficient and precision, and it slightly enhances its ROC area. This empirically supports the theoretical justification for ignoring constructors and access methods as discussed in Section 3.2. To the contrary, ignoring constructors and access methods in the computation of CC and SCOM weakens their performance in predicting faulty classes.
9. Considering transitive MMIs in the computation of LSCC, CC, and SCOM does not increase their standard coefficient, precision, recall, and ROC area. In other words, considering transitive MMIs for these metrics does not improve their ability in explaining the presence of faulty classes.

Table 7 also shows that the differences between the results obtained with and without accounting for transitive MMIs are very small and not statistically significant. Moreover, ignoring transitive MMIs even provides slightly better results. Accounting for transitive MMIs requires analyzing the source code to obtain information on method invocations, which requires additional work. As a result, from a fault prediction perspective, accounting for transitive MMIs does not bring clear benefits, whereas ignoring constructors and access methods improves the fault prediction results.

Let us now turn our attention to multivariate analysis and the impact of LSCC when building class fault-proneness prediction model based on MMI cohesion metrics. To study whether an optimal yet minimal multivariate model would contain LSCC, we used a backward selection process where all MMI metrics are first included in the model, and then removed one by one as long as one metric has a p-value above 0.05, starting with measures showing higher p-values. The results given in Table 8 show that LSCC, LCOM2, LCOM4, TCC, and DC_D remain in the prediction model as significant covariates. This somehow shows that they are complementary in predicting faults.

Metric	Std. Coeff.	Std. Error	p-value
LSCC	-0.97	0.15	< 0.0001
LCOM2	<0.01	<0.01	0.012
LCOM4	-0.15	0.07	0.041
TCC	-6.75	3.05	0.027
DC_D	6.31	3.06	0.039

Table 8: The model based on MMI metrics

To demonstrate that LSCC helps predict faulty classes, even when considering other MMI cohesion metrics, we performed multivariate regression analysis on four sets of MMI metrics including (1) CC_1, $SCOM_1$, and other MMI metrics excluding LSCC, (2) CC_1, $SCOM_1$, and other MMI metrics including LSCC, (3) CC_3, $SCOM_3$, and other MMI metrics excluding LSCC, and (4) CC_3, $SCOM_3$, and other MMI metrics including LSCC. The purpose of the analysis is to investigate whether including LSCC in the model improves its ability in predicting faulty classes in both cases when CC and SCOM are used as they are originally defined and when access methods and constructors are removed from their measurement. We applied the backward selection process explained earlier on each of the four sets of metrics and reported the results in Table 9. The results show that in both cases, when including or excluding access methods and constructors from the measurement of CC and SCOM, once LSCC is included in the set of selected metrics, it remains in the prediction model and the ability of the model in predicting faulty classes improves (Precision, recall, ROC). This is derived from comparing the results of the first and second, and third and fourth prediction models. The number of classes classified correctly as faulty and nonfaulty using each of the four considered models is reported in the last column of Table 9. These results show that the models that include LSCC classify faulty and nonfaulty classes more correctly than the ones that exclude LSCC, and therefore, these results confirm the results for precision, recall, and ROC area. In conclusion, the results show that LSCC is able to contribute to the prediction of faulty classes even when already accounting for other MMI cohesion metrics. Though our goal here is not to build fault prediction models, the results are useful in the sense that they show that LSCC captures complementary aspects of quality not accounted for by other MMI cohesion metrics.

Prediction Model	Remaining Metrics	Precision	Recall	ROC area	Correctly predicted classes
Model 1 excluding LSCC	$SCOM_1$, LCOM1, TCC, and DC_D	63.5	65.8	65.2	892
Model 2 including LSCC	LSCC, LCOM2, LCOM4, TCC, and DC_D	65.9	67.4	66.9	913
Model 3 excluding LSCC	CC_2, LCOM1, LCOM4, TCC, and DC_D	65.2	66.9	66.7	907
Model 4 including LSCC	LSCC, CC_2, $SCOM_2$, LCOM2, TCC, and DC_D	66.0	67.5	67.2	914

Table 9: Several models based on MMI metrics

Overall, the results show that MMI metrics that account for the degree of interaction between each pair of methods (namely, LSCC, CC, and SCOM) explain the presence of faulty classes more accurately than the other MMI metrics. This observation supports the hypothesis that metrics that account more precisely for interactions between methods in terms of number of shared attributes are preferable. Although LSCC, CC, and SCOM yield similar regression results in terms of fault prediction, LSCC is a better alternative because it also complies with important cohesion properties and, as shown by the multivariate analysis, it captures a complementary cohesion dimension. As a result, considering both the theoretical and empirical validation results together, LSCC is the best alternative among MMI metrics.

6.4. Threats to validity

A. External validity

Several factors may restrict the generality and limit the interpretation of our results. The first factor is that all four of the considered systems are implemented in Java. The second is that all the considered systems are open-source systems that may not be representative of all industrial domains, though this is common practice in the research community. Though differences in design quality and reliability between open source systems and industrial systems have been investigated (e.g., Samoladas et al. 2003, Samoladas et al. 2008, Spinellis et al. 2009), there is yet no clear, general result we can rely on. The third factor is that, though they are not artificial examples, the selected systems may not be representative in terms of the number and sizes of classes. To generalize the results, different systems written in different programming languages, selected from different domains, and including real-life, large-scale software should be taken into account in similar large-scale evaluations.

B. Internal validity

Though the presence of faults is one important aspect of quality, it is obviously not driven exclusively by class cohesion. Many other factors play an important role in driving the occurrence of faults (Arisholm et al. 2009). However, our goal here is not to predict faults in classes, but to investigate whether there is empirical evidence that LSCC is strongly related to observable aspects of quality, therefore suggesting that it is a well-defined cohesion measure, that is complementary or even a better option than existing MMI cohesion metrics. So, though the effect of cohesion on the presence of faults may be partly due to the correlation of cohesion with other unknown factors, it does not affect our objectives. Our cohesion measurement is an approximation because, as a practical necessity to enable statistical analysis, it is based on the latest version of the source code, that is the version after the faults are corrected. This likely affects the strength of the observed relationships between cohesion and fault occurrences. However, this is a quite common practice in similar research endeavors as mentioned earlier in Section 6.3.

7. Conclusions and Future Work

This paper introduces a new cohesion metric (LSCC) that addresses a number of problems in existing cohesion metrics that account for interactions between methods, an important category of cohesion metrics (MMI) to support refactoring. It is defined to be used during the Low-Level Design (LLD) phase of object-oriented software, and is based on measuring the degree of method pair interactions caused by the sharing of attributes in a class. The metric considers the number of shared attributes between each pair of methods and uses it to measure the degree of "similarity" between methods. The main targeted application of LSCC in this paper is to define an indicator to guide the refactoring of weakly cohesive classes. LSCC can easily account for transitive interactions and class inheritance for applications where it is deemed necessary. The effect of ignoring or considering constructors, destructors, and access methods in LSCC calculation is theoretically and empirically investigated. LSCC satisfies mathematical properties that have been widely considered necessary for cohesion measures. Empirical results from a large-scale study on four systems show that LSCC, considered individually or in combination with other MMI cohesion metrics, is one of the most strongly related, from a statistical standpoint, to fault occurrences in classes. In addition, the correlation study shows that LSCC captures a cohesion measurement dimension of its own. LSCC was shown on a case study to provide accurate guidance regarding the removal of methods from a class, an example refactoring operation. In summary, LSCC features the positive sides of other metrics and eliminates their problems. That is, except LSCC, none of the existing metrics fulfills all of the following criteria: (1) the metric is based on MMIs such that it can be used to guide refactoring activities, (2) the metric satisfies widely accepted cohesion properties, and (3) the metric has relatively strong fault prediction power. All other studied metrics in this paper are MMI-based, and therefore, all of them satisfy the first criterion. However, CC, SCOM, LCOM1, and LCOM2 do not satisfy the second criterion, TCC, LCC, DC_D, and DC_I do not satisfy the third one, and LCOM3 and LCOM4 do not satisfy both the second and third criteria. Taking into account both theoretical and empirical results, LSCC is therefore the best alternative cohesion metric during the LLD phase for the sake of supporting class refactoring.

Our metric does not distinguish between attributes and methods of different accessibility levels (i.e., public, private, and protected). Studies of the effect of considering or ignoring private and protected attributes and methods on the computation of LSCC and its fault prediction power are left open for future research. Finally, assessing empirically the impact of inheritance on LSCC and its fault prediction power is also relevant but requires complex static and dynamic analysis, as indicated in Section 6.1.There are several approaches to validate cohesion metrics. In this paper, we followed a widely used approach in which the cohesion is studied in terms of fault prediction capability, which is an indirect way to assess them as quality indicators. A future study could focus on applying other validation approaches such as providing the classes to developers to evaluate cohesion based on intuition and comparing the findings with LSCC results.

Acknowledgments
The authors would like to acknowledge the support of this work by Kuwait University Research Grant WI04/07. In addition, the authors would like to thank Walid Bahsow for developing the class cohesion measuring tool and Saqiba Sulman for assisting in collecting the cohesion results.

References

H. Abdi, Bonferroni and Sidak corrections for multiple comparisons, *Neil Salkind (ed.), Encyclopedia of Measurement and Statistics*, Thousand Oaks, CA: Sage, 2007, pp. 1-9.

K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, Investigating effect of design metrics on fault proneness in object-oriented systems, *Journal of Object Technology*, 6(10), 2007, pp. 127-141.

J. Al Dallal, Efficient program slicing algorithms for measuring functional cohesion and parallelism, *International Journal of Information Technology*, 4(2), 2007, pp. 93-100.

J. Al Dallal, Software similarity-based functional cohesion metric, *IET Software*, 2009, 3(1), pp. 46-57.

J. Al Dallal_a, Mathematical validation of object-oriented class cohesion metrics, *International Journal of Computers*, 4(2), 2010, pp. 45-52.

J. Al Dallal_b, Measuring the discriminative power of object-oriented class cohesion metrics, *IEEE Transactions on Software Engineering*, In press, 2010.

J. Al Dallal_c, Improving the applicability of object-oriented class cohesion metrics, submitted for publication in *IEEE Transactions on Software Engineering*, 2010.

J. Al Dallal and L. Briand, An object-oriented high-level design-based class cohesion metric, submitted for publication in *Information and Software Technology*, 2010.

M. Alshayeb, Empirical investigation of refactoring effect on software quality, *Information and Software Technology*, 51(9), 2009, 1319-1326.

E. Arisholm, L. C. Briand, and A. Foyen, Dynamic coupling measurement for object-oriented software, *IEEE Transaction on Software Engineering*, 30(8), 2004, pp. 491-506.

E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, accepted for publication on the *Journal of Systems and Software*, 2009.

L. Badri and M. Badri, A Proposal of a new class cohesion criterion: an empirical study, *Journal of Object Technology*, 3(4), 2004, pp. 145-159.

J. Bansiya, L. Etzkorn, C. Davis, and W. Li, A class cohesion metric for object-oriented designs, *Journal of Object-Oriented Program*, 11(8), 1999, pp. 47-52.

V. Barnett and T. Lewis, *Outliers in Statistical Data*, John Wiley and Sons, 3^{rd} e, 1994, pp. 584.

J. Bieman and B. Kang, Cohesion and reuse in an object-oriented system, *Proceedings of the 1995 Symposium on Software reusability*, Seattle, Washington, United States, 1995, pp. 259-262.

J. Bieman and B. Kang, Measuring design-level cohesion, *IEEE Transactions on Software Engineering*, 24(2), 1998, pp. 111-124.

J. Bieman and L. Ott, Measuring functional cohesion, *IEEE Transactions on Software Engineering*, 20(8), 1994, pp. 644-657.

C. Bonja and E. Kidanmariam, Metrics for class cohesion and similarity between methods, *Proceedings of the 44th Annual ACM Southeast Regional Conference*, Melbourne, Florida, 2006, pp. 91-95.

L. C. Briand, C. Bunse, and J. Daly, A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs, *IEEE Transactions on Software Engineering*, 27(6), 2001a, pp. 513-530.

L. C. Briand, J. Daly, and J. Wuest, A unified framework for cohesion measurement in object-oriented systems, *Empirical Software Engineering - An International Journal*, 3(1), 1998, pp. 65-117.

L. C. Briand , S. Morasca , and V. R. Basili, Defining and validating measures for object-based high-level design, *IEEE Transactions on Software Engineering*, 25(5), 1999, pp. 722-743.

L. C. Briand, J. Wust, J. Daly, and V. Porter, Exploring the relationship between design measures and software quality in object-oriented systems, *Journal of System and Software*, 51(3), 2000, pp. 245-273.

L. C. Briand and J. Wust, Empirical studies of quality models in object-oriented systems, *Advances in Computers*, Academic Press, 2002, pp. 97-166.

L. C. Briand, J. Wüst, and H. Lounis, Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs, *Empirical Software Engineering*, 6(1), 2001b, pp. 11-58.

H. S. Chae, Y. R. Kwon, and D. Bae, A cohesion measure for object-oriented classes, *Software—Practice & Experience*, 30(12), 2000, pp.1405-1431.

H. S. Chae, Y. R. Kwon, and D. Bae, Improving cohesion metrics for classes by considering dependent instance variables, *IEEE Transactions on Software Engineering*, 30(11), 2004, pp. 826-832.

Z. Chen, Y. Zhou, and B. Xu, A novel approach to measuring class cohesion based on dependence analysis, *Proceedings of the International Conference on Software Maintenance*, 2002, pp. 377-384.

S.R. Chidamber and C.F. Kemerer, Towards a Metrics Suite for Object-Oriented Design, *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Special Issue of SIGPLAN Notices, 26(10), 1991, pp. 197-211.

S.R. Chidamber and C.F. Kemerer, A Metrics suite for object Oriented Design, *IEEE Transactions on Software Engineering*, 20(6), 1994, pp. 476-493.

S. Counsell, S. Swift, and J. Crampton, The interpretation and utility of three cohesion metrics for object-oriented design, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(2), 2006, pp.123-149.

I. Czibula and G. Serban, Improving systems design using a clustering approach, *IJCSNS International Journal of Computer Science and Network Security*, 6(12), 2006, pp. 40-49.

G. Dunteman, *Principal Components Analysis*, Saga University Paper No. 7-69, Saga Publications, US, pp. 96.

T. Emerson, A discriminant metrics for module cohesion, *In Proceedings of the 7th International Conference on Software Engineering*, 1984, pp. 294-303.

L. Etzkorn, S. Gholston, J. Fortune, C. Stein, D. Utley, P. Farrington, and G. Cox, A comparison of cohesion metrics for object-oriented systems, *Information and Software Technology*, 46(10), 2004, pp. 677-687.

L. Fernández, and R. Peña, A sensitive metric of class cohesion, *International Journal of Information Theories and Applications*, 13(1), 2006, pp. 82-91.

GanttProject, http://sourceforge.net/projects/ganttproject/, August 2009

G. Gui and P. D. Scott, Coupling and cohesion measures for evaluation of component reusability, *International Conference on Software Engineering, Proceedings of the 2006 international workshop on Mining software repositories*, 2006, pp. 18-21.

T. Gyimothy, R. Ferenc, and I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, *IEEE Transactions on Software Engineering*, 3(10), 2005, pp. 897-910.

J. A. Hanley and B. J. McNeil, The meaning and use of the area under a receiver operating characteristic (ROC) curve, *Radiology*, 143(1), 1982, pp. 29-36.

B. Henderson-sellers, *Object-Oriented Metrics Measures of Complexity*, Prentice-Hall, 1996.

M. Hitz and B. Montazeri, Measuring coupling and cohesion in object oriented systems, *Proceedings of the International Symposium on Applied Corporate Computing*, 1995, pp. 25-27.

D. Hosmer and S. Lemeshow, *Applied Logistic Regression*, Wiley Interscience, 2000, 2nd edition.

Illusion, http://sourceforge.net/projects/aoi/, August 2009.

JabRef, http://sourceforge.net/projects/jabref/, August 2009

JHotDraw, http://sourceforge.net/projects/jhotdraw/, May 2010.

I. T. Jolliffe, *Pincipal Component Analysis*, Springer, 1986.

B. Kitchenham, S. L. Pfleeger, and N. Fenton, Towards a framework for software measurement validation, *IEEE Transactions on Software Engineering*, 21(12), 1995, pp. 929-944.

A. Lakhotia, Rule-based approach to computing module cohesion, *Proceedings of the 15th international conference on Software Engineering*, Baltimore, US, 1993, pp. 35-44.

W. Li and S.M. Henry, Maintenance metrics for the object oriented paradigm. *In Proceedings of 1st International Software Metrics Symposium*, Baltimore, MD, 1993, pp. 52-60.

A. De Lucia, R. Oliveto, and L. Vorraro, Using structural and semantic metrics to improve class cohesion, *In Proceedings of IEEE International Conference on Software Maintenance*, 2008, pp. 27-36.

A. Marcus, D. Poshyvanyk, and R. Ferenc, Using the conceptual cohesion of classes for fault prediction in object-oriented systems, *IEEE Transactions on Software Engineering*, 34(2), 2008, pp. 287-300.

T. Meyers and D. Binkley, An empirical study of slice-based cohesion and coupling metrics, *ACM Transactions on Software Engineering Methodology*, 17(1), 2007, pp. 2-27.

H. Olague, L. Etzkorn, S. Gholston, and S. Quattlebaum, Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes, *IEEE Transactions on Software Engineering*, 33(6), 2007, pp. 402-419.

D. Olson and D. Delen, *Advanced Data Mining Techniques*, Springer, 1st edition, 2008.

Openbravo, http://sourceforge.net/projects/openbravopos, August 2009.

L. Ott and J. Thuss, Slice based metrics for estimating cohesion, *Proceedings of the First International Software Metrics Symposium*, Baltimore, 1993, pp. 71-81.

I. Samoladas, S. Bibi, I. Stamelos, and G.L. Bleris. Exploring the quality of free/open source software: a case study on an ERP/CRM system, *9th Panhellenic Conference in Informatics*, Thessaloniki, Greece, 2003.

I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos, The SQO-OSS quality model: measurement based open source software evaluation, *Open Source Development, Communities and Quality*, 275, 2008, pp. 237-248.

S. Siegel and J. Castellan, *Nonparametric Statistics for the Behavioral Sciences*, McGraw-Hill, 2nd edition, 1988.

G. Snedecor and W. Cochran, *Statistical Methods*, Blackwell Publishing Limited, 1989, 8th edition.

D. Spinellis, G. Gousios, V. Karakoidas, P. Louridas, P. J. Adams, I. Samoladas, and I. Stamelos, Evaluating the quality of open source software, *Electronic Notes in Theoretical Computer Science*, 233, 2009, pp. 5-28.

R. Subramanyam and M. Krishnan, Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects, *IEEE Transactions on Software Engineering*, 29(4), 2003, pp. 297-310.

L. Tokuda and D. Batory, Evolving object-oriented designs with refactorings, *Automated Software Engineering*, 8, 2001, pp. 89-120.
D. Troy and S. Zweben, Measuring the quality of structured designs, *Journal of Systems and Software*, 2, 1981, pp. 113-120.
J. Wang, Y. Zhou, L. Wen, Y. Chen, H. Lu, and B. Xu, DMC: a more precise cohesion measure for classes. *Information and Software Technology*, 47(3), 2005, pp. 167-180.
E. Yourdon and L. Constantine, *Structured Design*, Prentice-Hall, Englewood Cliffs, 1979.
Y. Zhou, J. Lu, H. Lu, and B. Xu, A comparative study of graph theory-based class cohesion measures, ACM SIGSOFT Software Engineering Notes, 29(2), 2004, pp. 13-13.
Y. Zhou, B. Xu, J. Zhao, and H. Yang, ICBMC: an improved cohesion measure for classes, *Proc. of International Conference on Software Maintenance*, 2002, pp. 44-53.