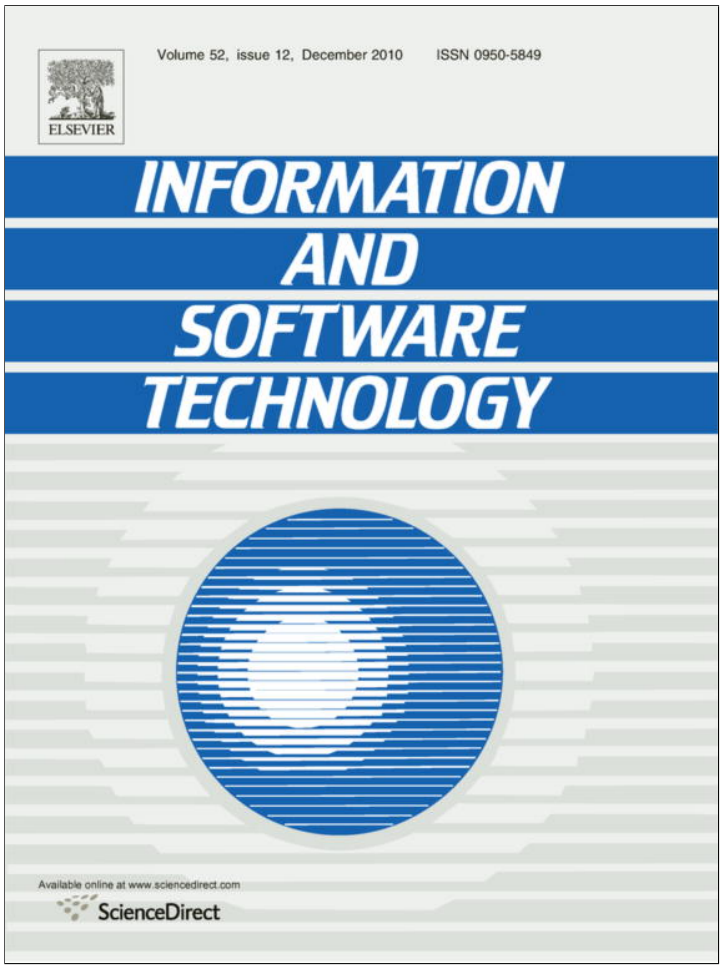

Information and Software Technology 52 (2010) 1346–1361

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

# An object-oriented high-level design-based class cohesion metric

Jehad Al Dallal [a,*], Lionel C. Briand [b,c]

[a] Department of Information Science, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait
[b] Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway
[c] Department of Informatics, University of Oslo, Norway

ARTICLE INFO

*Article history:*
Received 25 April 2010
Received in revised form 2 July 2010
Accepted 26 August 2010
Available online 19 September 2010

*Keywords:*
Object-oriented software quality
Object-oriented design
Class cohesion
Fault prediction

ABSTRACT

*Context:* Class cohesion is an important object-oriented software quality attribute. Assessing class cohesion during the object-oriented design phase is one important way to obtain more comprehensible and maintainable software. In practice, assessing and controlling cohesion in large systems implies measuring it automatically. One issue with the few existing cohesion metrics targeted at the high-level design phase is that they are not based on realistic assumptions and do not fulfill expected mathematical properties.
*Objective:* This paper proposes a High-Level Design (HLD) class cohesion metric, which is based on realistic assumptions, complies with expected mathematical properties, and can be used to automatically assess design quality at early stages using UML diagrams.
*Method:* The notion of similarity between pairs of methods and pairs of attribute types in a class is introduced and used as a basis to introduce a novel high-level design class cohesion metric. The metric considers method–method, attribute–attribute, and attribute–method direct and transitive interactions. We validate this Similarity-based Class Cohesion (SCC) metric theoretically and empirically. The former includes a careful study of the mathematical properties of the metric whereas the latter investigates, using four open source software systems and 10 cohesion metrics, whether SCC is based on realistic assumptions and whether it better explains the presence of faults, from a statistical standpoint, than other comparable cohesion metrics, considered individually or in combination.
*Results:* Results confirm that SCC is based on clearly justified theoretical principles, relies on realistic assumptions, and is an early indicator of quality (fault occurrences).
*Conclusion:* It is concluded that SCC is both theoretically valid and supported by empirical evidence. It is a better alternative to measure class cohesion than existing HLD class cohesion metrics.


## 1. Introduction

Software engineering aims at developing techniques and tools to promote quality software that is stable and easy to maintain. In order to assess and improve software quality during the development process, developers and managers use, among other means, ways to automatically measure the software design. To this aim, many metrics have been proposed to estimate different attributes such as cohesion, coupling, and complexity [1].

The cohesion of a module is one important property of software design and refers to the relatedness of software module constituents. A highly cohesive module has one basic function and cannot be split into separate modules [2]. Highly cohesive modules are believed to be more understandable, modifiable, and maintainable than less cohesive modules [3,4].

Over the last decade, object-oriented programming languages, such as C++ and Java, have gained popularity in the software industry. In the object-oriented paradigm, classes serve as the basic units of design. The constituents or members of a class are its attributes and methods. Therefore, class cohesion refers to the relatedness of the class members. Three possible types of related interactions are defined including method–method, method–attribute, and attribute–attribute interactions. The method–method interaction between a pair of methods is defined when both methods access a common attribute, or when one method invokes the other one. The method–attribute interaction between a method and an attribute is defined when the method accesses the attribute. Finally, the attribute–attribute interaction between a pair of attributes is defined when both attributes are accessed by a common method.

Several class cohesion metrics have been proposed in the literature. These metrics can be applicable based on High-Level Design (HLD) or Low-Level Design (LLD) information. HLD class cohesion metrics rely on information related to class and method interfaces [5–7]. The more numerous LLD class cohesion metrics, such as

* Corresponding author. Tel.: +965 6173888.
*E-mail addresses:* jehad@cfw.kuniv.edu (J. Al Dallal), briand@simula.no (L.C. Briand).

0950-5849/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.08.006

those proposed by Chidamber and Kemerer [8], Chidamber and Kemerer [9], Hitz and Montazeri [10], Bieman and Kang [11], Chen et al. [4], Badri and Badri [12], Wang et al. [13], and Fernandez and Pena [14], require analyzing the algorithms used in the class methods (or the code itself if available) or access to highly precise method postconditions. The LLD cohesion metrics use finer-grained information than that used by HLD cohesion metrics. That is, based on the LLD, all method–method, method–attribute, and attribute–attribute interactions can be precisely defined. On the other hand, one advantage of HLD class cohesion metrics is that they identify potential cohesion issues early, during the HLD phase. Detecting class cohesion issues, and correcting the corresponding class artifacts later (during the LLD or implementation phase), is much more costly than performing the same tasks early (during the HLD phase). Improving class cohesion during the HLD phase saves development time, reduces development costs, and increases overall software quality [7].

The HLD class cohesion metrics proposed to date have several drawbacks. First, some of them are based on assumptions that are yet to be empirically validated and are probably unrealistic. For instance, in some of the seminal work on the topic [6,7,15], the assumption is that the types of the method parameters match the types of the attributes accessed by the method. However, this assumption has little empirical support beyond the study of Counsell et al. [7] based on 21 C++ classes. Second, some key features of object-oriented programming languages, such as inheritance, are not considered in HLD class cohesion metrics proposed to date. Third, certain proposed metrics, such as Cohesion Among Methods in a Class (CAMC) [6] and Normalized Hamming Distance (NHD) [7,15], have not been validated in terms of their mathematical properties and in fact violate key properties. Fourth, some metrics ignore transitive interactions and method-invocation interactions. Finally, research in the area of HLD class cohesion measurement needs more, larger scale empirical studies that examine the correlation among the proposed metrics and that explore the relationships between HLD cohesion and software quality.

In this paper, we review and discuss some recently proposed design class cohesion metrics, with an emphasis on HLD metrics. We introduce a new model to predict the relationship between parameter types and attribute types. To do so, we define the notion of similarity between a pair of methods and a pair of attribute types and use this as a basis to measure class cohesion. We thus introduce a novel HLD class cohesion metric that we refer to as Similarity-based Class Cohesion (SCC). This metric accounts for method–method, method–attribute, attribute–attribute, and method invocation direct and transitive interactions identified using the class and communication diagrams defined by the Unified Modeling Language (UML), which are commonly used for describing object-oriented designs. Our expectations, in terms of data available in HLD UML diagrams, are consistent with main stream object-oriented, UML-based methodologies (e.g., [16–18]).

The method implementations and, consequently, the information about the method–attribute interactions are not available during the HLD phase. Our metric is based on the assumption that the matching between the method parameter types and the attribute types capture most of the method–attribute interactions. We investigated our assumption versus the assumptions proposed in some of the seminal proposals on the topic, on which our work is based [6,7,15]. The results show that our assumption captures the method–attribute interactions more precisely than the other assumptions.

The validity of a metric has to be studied both theoretically and empirically [19]. The theoretical validation tests whether the proposed metric complies with the necessary properties of the measured attribute. The empirical validation tests whether measured and predicted values are consistent with each other.

Consistent with this general validation approach, SCC is then validated from both theoretical and empirical standpoints. Our theoretical validation involves analyzing the compliance of SCC with the properties proposed by Briand et al. [20]. The empirical validation involves applying 10 cohesion metrics, including the most common cohesion metrics in the literature and SCC, to classes selected from four open source Java systems. We explore the correlations between the 10 considered metrics, thus determining whether SCC captures new information, and study the fault-prediction power of the metrics considered individually and in various combinations. To perform this study, we collected fault data for the classes in the considered software systems from publicly available fault repositories and statistically analyzed the relationship between cohesion values and the presence of faults in classes. The results show that SCC defines a cohesion dimension of its own, is based on assumptions that are more realistic than other similar metrics, and has relatively high fault-prediction power compared to other proposed metrics. Though building fault-prediction models is not an objective of this paper, as further discussed below, this is used as a way to gather empirical evidence that SCC better relates to class quality than other, comparable metrics.

As a result, theoretical and empirical validation results show that SCC is a better HLD class cohesion metric than the existing alternatives. Note that the existing HLD class cohesion metrics are either theoretically invalid or lack empirical support. SCC uses information provided in UML diagrams, and therefore, can be incorporated into existing UML design tools to predict the cohesion of individual classes as the UML diagrams are being drawn. The metric is dedicated to predict cohesion among members of an individual class. Other existing metrics can be applied to predict the cohesion of an individual method (e.g., [21]), at a lower level, or a component including a set of related classes (e.g., [22]), at a higher level. The proposed metric predicts potential class cohesion problems early during HLD and points to weakly designed classes that may need to be refactored. Corrections during HLD are expected to be much less expensive than performing the same tasks during the LLD or implementation phases.

This paper is organized as follows. Section 2 reviews related work. Section 3 defines the model used by our proposed metric. In Section 4, we define the SCC metric, and in Section 5, we validate it theoretically. Section 6 illustrates several empirical case studies and reports and discusses their results. Finally, Section 7 concludes and discusses future work.

## 2. Related work

In this section, we summarize a widely used set of mathematical properties that all class cohesion metrics should satisfy. In addition, we review and discuss several existing design class cohesion metrics for object-oriented systems and other related work in the area of measuring software cohesion. Finally, we provide an overview of two related UML diagrams to demonstrate that the information we rely on is available in high-level designs with UML.

### 2.1. A class cohesion metric's necessary properties

Briand et al. [20] define four mathematical properties that provide a supportive underlying theory for class cohesion metrics. The first property, called *non-negativity* and *normalization*, is that the 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) and the cohesion is equal to Max if all possible interactions within the class are present. The third property, called *monotonicity*, holds that adding cohesive interactions to the module cannot decrease its cohesion. The fourth property, called *cohesive modules*, holds that merging 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′) $\leqslant$ max {cohesion(c1), cohesion(c2)}. If a metric does not satisfy any of these properties, it is considered ill-defined [20]. Despite its widespread use and acceptance, many research studies do not report such theoretical validation for cohesion metrics or any equivalent alternative [23]. In this paper, we theoretically validate our proposed class cohesion metric using the properties introduced by Briand et al. [20]. These properties are specific to class cohesion metrics and have been widely used, whereas other theoretical validation properties proposed in the literature (e.g., [19]) are of more general nature.

### 2.2. Design class cohesion metrics

Several metrics have been proposed in the literature to measure class cohesion during the system HLD and LLD phases. These metrics use different underlying models and different formulas. We start by discussing in detail two of the most important proposals regarding HLD cohesion metrics and then briefly discuss LLD metrics and other less directly relevant work.

#### 2.2.1. Cohesion among methods in a class

Bansiya et al. [6] propose a design class cohesion metric called Cohesion Among Methods in a Class (CAMC). The CAMC metric uses a parameter-occurrence matrix that has a row for each method and a column for each data type that appears 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 the matrix is 1 when the $i$th method has a parameter of the $j$th data type and is 0 otherwise. In the matrix, the class type is always included in the parameter type list, and every method interacts with this data type because every method implicitly has an identity parameter. This means that one of the columns is filled entirely with 1s. The CAMC metric is defined as the ratio of the total number of 1s in the matrix to the total size of the matrix. As per our knowledge, the CAMC metric was the first proposed HLD class cohesion metric and is based on information available in UML class diagrams [24]. However, several related issues are left open for further research including the consideration of class inheritance and transitive interactions. In addition, the CAMC metric is based on assumptions for which there is little empirical evidence: the types of the method parameters are expected to match the types of the attributes accessed by the method. Among the different types of cohesive interactions, defined in Section 1, the CAMC metric considers only attribute–method interactions. Finally, the metric does not satisfy the normalization property because each parameter type is used by at least one method and one of the parameter types is used by all methods. Therefore, the minimum number of 1s in the matrix is $k + l - 1$, where $k$ is the number of rows and $l$ is the number of columns. In this case, $\text{CAMC}_{\text{min}} = (k + l - 1)/kl$. Consequently, the normalized CAMC (NCAMC) equals $(k \cdot l \cdot \text{CAMC} - k - l + 1)/(k \cdot l - k - l + 1)$. Counsell et al. [7] suggest omitting the type of class from the parameter-occurrence matrix and calculating CAMC from the modified matrix. We refer to this metric as $\text{CAMC}_{\text{c}}$. In this case, the minimum number of 1s in the matrix is $l$, so $\text{CAMC}_{\text{cmin}} = l/kl = 1/k$. Consequently, the normalized $\text{CAMC}_{\text{c}}$ equals $(k \cdot \text{CAMC}_{\text{c}} - 1)/(k - 1)$. Given a parameter-occurrence matrix without the class type and $a = \sum_{i=1}^{k}\sum_{j=1}^{l} c_{ij}$, where $c_{ij}$ is the value at row $i$ and column $j$ in the matrix, CAMC can be calculated as follows [7]:

$$\text{CAMC} = \frac{a + k}{k(l + 1)} \tag{1}$$

#### 2.2.2. Normalized Hamming Distance (NHD) metric

Counsell et al. [7] propose and discuss the interpretation and utility of a design class cohesion metric called the Normalized Hamming Distance (NHD). This metric uses the same parameter-occurrence matrix as the CAMC metric (the type of class is not considered). This approach 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. Formally, the metric is defined as follows:

$$\text{NHD} = \frac{2}{lk(k-1)}\sum_{j=1}^{k-1}\sum_{i=j+1}^{k} a_{ij} = 1 - \frac{2}{lk(k-1)}\sum_{j=1}^{l} x_j(k - x_j) \tag{2}$$

where $a_{ij}$ is the number of entries in rows $i$ and $j$ for which both are 1, and $x_j$ is the number of 1s in the $j$th column of the parameter-occurrence matrix. The metric has several limitations. The first is that it is counter-intuitive to consider the absence of a parameter type in a pair of methods to be a cohesive relation: a pair of methods would be considered fully cohesive if they did not have any parameters. This limitation can be overcome by ignoring this case when accounting for cohesion. The second limitation is that the metric satisfies the normalization property only when the class has two methods. The third and fourth limitations are that it does not satisfy the monotonicity and cohesive module properties unless the Hamming Distance definition is modified which implies introducing a completely different metric. In other words, NHD does not satisfy most of the necessary properties proposed by Briand et al. [20] for class cohesion metrics. Finally, NHD does not consider class inheritance, transitive interactions, and the types of cohesive interactions defined in Section 1, other than method–method interactions. Scaled NHD (SNHD) is a metric that represents the closeness of the NHD metric to the maximum value of NHD compared to the minimum value [7], and therefore, it can be used as a basis for normalizing NHD. The SNHD value ranges within the interval $[-1, 1]$, where the closer the value is to zero, the less cohesive is the class.

### 2.3. Overview of other relevant work

Yourdon et al. [25] 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 are used to measure the cohesion of procedural programs, such as the control flow graph [26], the variable dependence graph [27], and program data slices [2,21,28–30]. Cohesion has also been measured indirectly by examining the quality of the structured designs [31,32].

Several LLD class cohesion metrics have been proposed in the literature. These metrics are based on the use or sharing (i.e., common use) of class instance variables. The Lack of Cohesion of Methods (LCOM1) metric [8] counts the number of pairs of methods that do not share instance variables. Chidamber and Kemerer [9] proposed another version of the LCOM metric, referred to here as LCOM2, which calculates the difference between the number of method pairs that do and do not share instance variables. Li and Henry [33] use an undirected graph that represents each method as a node and the sharing of at least one instance variable as an edge. Class cohesion, LCOM3, is measured in terms of the number of connected components in the graph. This class cohesion was ex-

tended by Hitz and Montazeri [10], who added an edge between a pair of methods if one invokes the other. Bieman and Kang [11] proposed two class cohesion metrics, Tight Class Cohesion (TCC), which measures the relative number of directly connected pairs of methods, and Loose Class Cohesion (LCC), which measures the relative number of directly or transitively connected pairs of methods. These two metrics consider two methods to be connected if they share at least one instance variable or one of the methods invokes the other. The cohesion metric Degree of Cohesion ($DC_D$) is similar to TCC, and the metric $DC_I$ is similar to LCC [12], but they also consider two methods connected when they invoke the same method. Briand et al. [20] proposed a cohesion metric, called Coh, that computes the cohesion as the ratio of the number of distinct attributes accessed in the methods of a class. Wang et al. [13] 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. Fernandez and Pena [14] propose class cohesion metrics that consider the cardinality of intersection between each pair of methods. Henderson-Sellers [34] and Briand et al. [20] propose class cohesion metrics similar to CAMC but for the method/instance variable matrix. Chen et al. [4] use dependence analysis to explore attribute-attribute, attribute–method, and method–method interactions. They measure cohesion as the relative number of interactions. Al Dallal and Briand [35] propose a metric to measure cohesion during LLD. The metric is based on measuring the degree of similarity between each pair of methods in terms of the number of shared attributes. Al Dallal [36] proposes a distance-based HLD cohesion metric and discusses its sensitivity to changes in the class cohesive interactions. The metric is based on information available in the UML class diagram. This paper is extended here by (1) proposing the similarity-based metric, which measures the cohesion more directly than the distance-based metric, (2) accounting for class inheritance, (3) accounting for transitive class interactions, (4) empirically validating the underlying assumptions, and (5) validating the metric theoretically and empirically.

### 2.4. UML class and communications diagrams

UML is a standard language used for modeling object-oriented design. UML 2.0 [37,38] consists of 13 types of diagrams. In this paper, we are interested in class and communication diagrams, as they are typically part of high-level designs [16–18]. The class diagram describes the system's classes and the static relationships between them. The description of a class includes the names and types of the attributes and the names, return types, and parameter types of the methods. Fig. 1(a) shows a sample class diagram for the *AccountDialog* class.

The communication diagram describes the message (e.g., method call) flow between system's objects. Each object is represented by a box that includes the class name and the object name (optional). Messages between the objects are associated with links. Many messages can flow along a link. Each message is assigned to a small arrow indicating its direction and a sequence number. Invoking a method from another method belonging to the same class is modeled in the communication diagram by initiating both messages from the same object and prepending the sequence number of the invoked message with the sequence number of the invoking message. For example, the partial communication diagram shown in Fig. 1(b) indicates that the *showInfo* method invokes the *showExtraInfo* method, which in turn invokes the *showAddress* method. Note that our proposed metric only requires the availability of class and communication diagrams related to the considered classes. Other types of UML diagrams are not necessary. In addition, to predict the cohesion of a specific class, complete UML design diagrams for all classes are not required: Our proposed metric is applied on each class individually.

Genero et al. [24] provide a survey for object-oriented quality metrics based on UML diagrams. They show that the information required to build the parameter-occurrence matrix, used in the CAMC and NHD metrics, is available in the UML class diagram.

In summary, several class cohesion metrics are introduced in the literature. Few of these metrics are based on information available during HLD phase. The HLD cohesion metrics introduced to date do not consider transitive interactions and inheritance and they are based on models built on assumptions that are yet to be empirically validated. In addition, they are not validated theoretically against necessary class cohesion properties, and empirically against external quality attributes such as fault occurrences. In this paper, to address the above issues, we introduce Similarity-based Class Cohesion (SCC), a HLD class cohesion metric based on assumptions that we empirically investigate and that are shown to be realistic in several systems. The SCC metric can easily account for class inheritance and direct and transitive interactions. In addition, the SCC metric complies with widely accepted class cohesion properties and has relatively high fault-prediction power when compared to other proposed HLD and LLD cohesion metrics, thus providing indirect, empirical evidence that it might be a more meaningful measure of class cohesion than other existing class cohesion metrics.

## 3. Model definition

The similarity-based class cohesion metric introduced in this paper considers attribute–attribute, method–method, and attribute–method direct and transitive interactions. We introduce four types of matrices: (1) the Direct Method Invocation (DMI) matrix

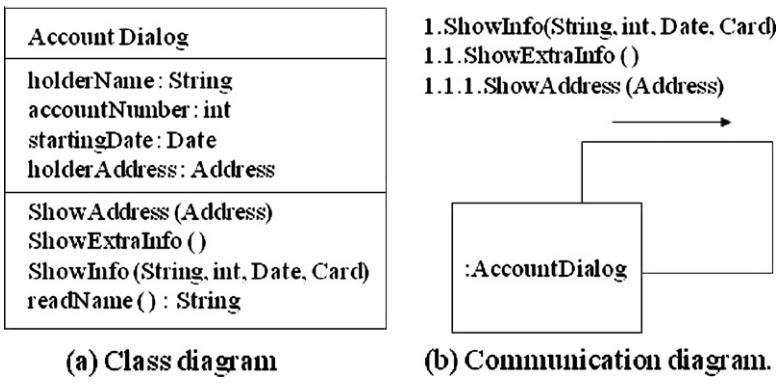


**Fig. 1.** UML class and communication diagrams for *AccountDialog*.

to model direct method-invocation interactions, (2) the Method Invocation (MI) matrix to model direct and transitive method-invocation interactions, (3) the Direct Attribute Type (DAT) matrix to model direct method–method, attribute–attribute, and attribute–method interactions, and (4) the Attribute Type (AT) matrix to model direct and transitive interactions modeled in the DAT matrix. The information source in UML diagrams relied upon for building these matrices will be identified.

### 3.1. Direct Method Invocation (DMI) matrix

Direct method-invocation interactions are explicitly defined in the UML communication diagram. These interactions are obtained and reported in a matrix called the DMI matrix. This matrix is a square binary $k \times k$ matrix, where $k$ is the number of methods in the class of interest. To construct the matrix, the names of the class methods are obtained from the UML class diagram reviewed in Section 2.4. The rows and columns of the DMI matrix are indexed by the methods, and for $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant k$,

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

A binary value of 1 in the DMI matrix indicates a cohesive direct method-invocation interaction.

Given the class diagram and communication diagrams shown in Fig. 1 for the *AccountDialog* class, the DMI matrix shown in Fig. 2 is constructed. The matrix shows that the *showInfo* method invokes directly the *showExtraInfo* method and the *showExtraInfo* method invokes directly the *showAddress* method. Note that there are other possible mechanisms for method invocations beyond simple method calls, such as the receipt of a signal in an active class (e.g., implemented in a *run()* method in Java). The later occurs in concurrent software with asynchronous communication between objects.

To account for class inheritance, all accessible methods inherited directly or indirectly have to be included in the DMI matrix. The inherited methods can be extracted from the UML class diagram.

### 3.2. Method Invocation (MI) matrix

The MI matrix is a square binary $k \times k$ matrix, where $k$ is the number of methods in the class of interest. The matrix models the direct and transitive method-invocation interactions and is derived from the DMI matrix. The rows and columns of the MI matrix are indexed by the methods, and for $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant k$,

$$m_{ij} = \begin{cases} 1 & \text{if the } i\text{th method invokes directly or transitively} \\ & \quad \text{the } j\text{th method,} \\ 0 & \text{otherwise} \end{cases}$$

| | showInfo | showAddress | showExtraInfo | readName |
|---|---|---|---|---|
| showInfo | 0 | 0 | 1 | 0 |
| showAddress | 0 | 0 | 0 | 0 |
| showExtraInfo | 0 | 1 | 0 | 0 |
| readName | 0 | 0 | 0 | 0 |

**Fig. 2.** The DMI matrix for the *AccountDialog* class.

| | showInfo | showAddress | showExtraInfo | readName |
|---|---|---|---|---|
| showInfo | 0 | 1 | 1 | 0 |
| showAddress | 0 | 0 | 0 | 0 |
| showExtraInfo | 0 | 1 | 0 | 0 |
| readName | 0 | 0 | 0 | 0 |

**Fig. 3.** The MI matrix for the *AccountDialog* class.

A binary value of 1 in the MI matrix indicates a cohesive direct or transitive method-invocation interaction. Fig. 3 shows the MI matrix for the *AccountDialog* class. The matrix shows that the *showInfo* method invokes directly or transitively the *showAddress* and the *showExtraInfo* methods and the *showExtraInfo* method invokes directly or transitively the *showAddress* method.

### 3.3. Direct Attribute Type (DAT) matrix

Bansiya et al. [6] and Counsell et al. [7] used the parameter-occurrence matrix for parameter types as the basis for their metrics. This matrix is based on the assumption that the set of attribute types accessed by a method is the intersection of this method's parameter types and the set of parameter types of all methods in the class. This assumption leads to two problems. The first problem is that some methods can have parameters of types that do not match the types of the attributes. In this case, methods that share these types are considered cohesive despite the fact that they do not share any attributes. The second problem is that, the parameter-occurrence matrix does not indicate whether all attributes are actually used within the methods. Therefore, in some cases, the class is considered fully cohesive despite the fact that some of its attributes are never used by the methods. Fig. 4(a) depicts these two problems by showing an extreme case in which all methods share all parameter types that do not match any of the attribute types and none of the attribute types match the method parameter types. In this example, we assume having a loosely cohesive class *classA* including two attributes and two methods, such that none of the attributes is used in the methods and the two methods are unrelated. Fig. 4(a) shows the corresponding class diagram. The two methods share the same parameter types: *int* and *String*. None of these types match any of the types of the attributes (*boolean* and *double*). In addition, the types of the two attributes are *boolean* and *double*, and none of these types match any of the parameter types. Fig. 4(b) shows the corresponding parameter-occurrence matrix. The matrix wrongly predicts the class to be fully cohesive because it shows that each of the two methods uses each of the parameter types, whereas none of the attributes is used in the methods.

Since the aim is to predict the sharing of attributes between the methods, we address the above two problems by introducing the DAT matrix. This matrix depicts the use of the types of the attributes themselves instead of depicting the use of the types of the method parameters. The matrix is a binary $k \times l$ matrix, where $k$ is the number of methods and $l$ is the number of distinct attribute types in the class of interest. The matrix is based on the assumption that the set of attribute types accessed by a method is the intersection of the set of this method's parameter types and the set of its class attribute types. To construct the matrix, the names and return types of the methods and the types of parameters and attributes are extracted from the UML class diagram reviewed in

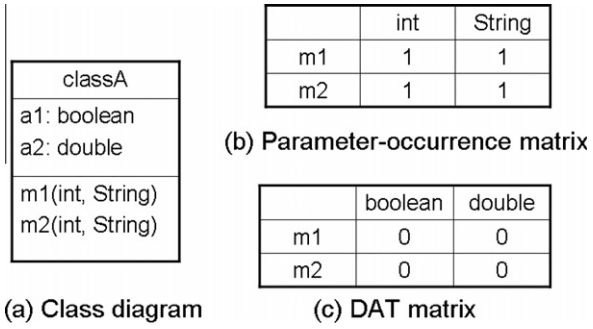


**Fig. 4.** A sample class diagram and its corresponding parameter-occurrence and DAT matrices.

Section 2.4. The DAT matrix has rows indexed by the methods and columns indexed by the distinct attribute types, and for $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant l$,

$$m_{ij} = \begin{cases} 1 & \text{if the } j\text{th attribute type is a type of at least one of the paramters or return of the } i\text{th method,} \\ 0 & \text{otherwise} \end{cases}$$

The DAT matrix solves the above two problems as follows. First, the DAT matrix ignores the parameter types that do not match any of the attribute types. This eliminates the possibility of wrongly predicting the method accessibility of non-existing attributes. Second, the DAT matrix shows all the attribute types including the non-accessed ones. This eliminates the possibility of wrongly predicting that a class whose attributes are not accessed by the methods to be highly cohesive. As opposed to the parameter-occurrence matrix, the DAT matrix given in Fig. 4(c) depicts the fact that the class has no attribute–method interactions (i.e., none of the attributes is accessed by any of the methods).

Based on a number of assumptions to be verified, the matrix is built using heuristics in order to provide information about likely interactions that are not directly visible at the HLD stage. Such heuristics will be verified in the case study section (Section 6.2). The matrix explicitly models direct attribute–method interactions. It is assumed that a method has a cohesive interaction with an attribute if the attribute type matches the type of at least one parameter or return value of the method. In addition, the matrix implicitly models method–method and attribute–attribute interactions. Based on our assumption, potentially, a method has a cohesive interaction with another method if their parameters or return values share the same attribute type. In addition, potentially, an attribute has a cohesive interaction with another attribute if their types are shared by a method. This indicates that the method potentially causes an interaction between the two attributes. A binary value of 1 in the DAT matrix indicates a cohesive attribute–method interaction. A cohesive method–method interaction is represented in the DAT by two rows that share binary values of 1 in a column. Similarly, a cohesive attribute–attribute interaction is represented in the DAT by two columns sharing binary values of 1 in a row. This matrix considers the return type of the method since it is possible that some methods access attributes not passed as parameters and return results that match the types of the accessed attributes. Consequently, the return type gives an indication of the accessed attributes within methods, and therefore, it should also be considered in the class cohesion metric.

Fig. 5 shows the DAT matrix of the *AccountDialog* class. The matrix shows that three of the attribute types are used by the *showInfo* method, one of the attribute types is used by the *showAddress* method, and one of the attributes is used by the *readName* method (as a return type). In addition, the matrix shows that the *showInfo* and *readName* methods share an attribute type, and the *String* attribute type is shared between two methods. The DAT matrix does not include the parameter type Card because it does not match any of the attribute types.

To consider class inheritance, all inherited methods included in the DMI matrix have to be considered in the DAT matrix as well. In addition, the distinct types of all directly and transitively accessible attributes have to be considered in the DAT matrix. These attribute types can be extracted from the UML class diagram.

| | String | int | Date | Address |
|---|---|---|---|---|
| showInfo | 1 | 1 | 1 | 0 |
| showAddress | 0 | 0 | 0 | 1 |
| showExtraInfo | 0 | 0 | 0 | 0 |
| readName | 1 | 0 | 0 | 0 |

**Fig. 5.** The DAT matrix for the *AccountDialog* class.

**Algorithm**: Constructing AT matrix
**Input**: DAT and MI matrices
**Output**: AT matrix
**Steps**:
*for* i=1 to no. of rows in DAT matrix
  *for* j=1 to no. of columns in DAT matrix
    AT[i,j]=DAT[i,j]
*for* i=1 to no. of rows in IMI matrix
  *for* j=1 to no. of columns in MI matrix
    *if* (i j AND MI[i,j]=1) *then*
      *for* k=1 to no. of columns in DAT matrix
        AT[i,k]=DAT[i,k] OR DAT[j,k]

**Fig. 6.** Constructing the AT matrix algorithm.

### 3.4. Attribute Type (AT) matrix

The AT matrix is similar to the DAT matrix in the sense that it has the same rows and columns, but it differs in that it models both direct and transitive interactions. A value of 1 in the matrix indicates that the attribute type matches the parameter type of the method or the methods directly or transitively invoked by the method. The AT can be generated from both DAT and MI matrices by applying the algorithm given in Fig. 6. The algorithm modifies the DAT matrix by replacing the row of the caller by the result of applying the logic OR for each corresponding cell in the rows representing the call originator and call recipient. Fig. 7 shows the AT matrix constructed using the algorithm in Fig. 6. The matrix is similar to the DAT matrix except that the *showInfo* and *showExtraInfo* methods use the *Address* attribute type transitively.

## 4. The Similarity-based Class Cohesion (SCC) metric definition

The SCC metric uses the AT matrix to measure the method–method interactions caused by sharing attribute types, the attribute-attribute interactions caused by the expected use of attribute within the methods, and the attribute–method interactions. In addition, the SCC metric uses the MI matrix to measure the method–method interactions caused by method invocations. The cohesions caused by the four types of interactions are referred to as Method–Method through Attributes Cohesion (MMAC), Attribute–Attribute Cohesion (AAC), Attribute–Method Cohesion (AMC), and Method–Method Invocation Cohesion (MMIC), respectively. The SCC metric uses the AT and MI matrices, and it therefore considers both direct and transitive interactions.

### 4.1. MMAC and AAC metrics

The similarity between two items is the collection of their shared properties. In the context of the AT matrix, introduced in Section 3, the similarity between two rows and two columns quantifies the cohesion between a pair of methods and a pair of attributes, respectively. The similarity between a pair of rows or columns is defined as the number of entries in a row or column that have the same binary values of "1" as the corresponding elements in the other row or column. The normalized similarity, denoted as $ns(i,j)$, between a pair of rows or columns $i$ and $j$ is defined as the ratio of similarity between the two rows or columns

| | String | int | Date | Address |
|---|---|---|---|---|
| showInfo | 1 | 1 | 1 | 1 |
| showAddress | 0 | 0 | 0 | 1 |
| showExtraInfo | 0 | 0 | 0 | 1 |
| readName | 1 | 0 | 0 | 0 |

**Fig. 7.** The AT matrix for the *AccountDialog* class.

to the number of entities Y in the row or column of the matrix, and it is defined formally as follows:

$$ns(i,j) = \frac{\sum_{x=1}^{Y}(m_{ix} \wedge m_{jx})}{Y}, \tag{3}$$

where $\wedge$ is the logical and relation.

Cohesion refers to the degree of similarity between module components. The MMAC is the average cohesion of all pairs of methods and the AAC is the average cohesion of all pairs of attributes. Formally, using the AT matrix, the MMAC of a class C, consisting of $k$ methods and $l$ distinct attribute types, is defined formally as follows:

$$\mathrm{MMAC(C)} = \begin{cases} 0 & \text{if } k = 0 \text{ or } l = 0 \\ 1 & \text{if } k = 1, \\ \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} ns(i,j) & \text{otherwise.} \end{cases} \tag{4}$$

By substituting Formula (3) into Formula (4), the MMAC of class C is calculated in the case of a class with multiple methods as follows:

$$\mathrm{MMAC(C)} = \frac{2}{lk(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{w=1}^{l} (m_{iw} \wedge m_{jw}) \tag{5}$$

The following metric is an alternative form of the MMAC metric, which obtains the same value, facilitates the analysis of the metric, and speeds up its computation:

**Proposition 4.1.** For any class C,

$$\mathrm{MMAC(C)} = \begin{cases} 0 & \text{if } k = 0 \text{ or } l = 0 \\ 1 & \text{if } k = 1, \\ \frac{\sum_{i=1}^{l} x_i(x_i-1)}{lk(k-1)} & \text{otherwise} \end{cases} \tag{6}$$

where $x_i$ is the number of 1s in the $i$th column of the AT matrix.

**Proof.** By definition, when $k = 1$ or $k = 0$ and $l = 0$, Eqs. (5) and (6) are equal. Otherwise, for the $i$th column, there are $x_i(x_i - 1)/2$ similarities between the methods, and therefore,

$$\begin{aligned} \mathrm{MMAC(C)} &= \frac{2}{lk(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{w=1}^{l} (m_{iw} \wedge m_{jw}) \\ &= \frac{2}{lk(k-1)} \frac{\sum_{i=1}^{l} x_i(x_i-1)}{2}, \end{aligned}$$

which equals the above formula. □

Similarly, the AAC of a class C is defined formally as follows:

$$\mathrm{AAC(C)} = \begin{cases} 0 & \text{if } k = 0, \text{ or } l = 0, \\ 1 & \text{if } l = 1, \\ \frac{\sum_{i=1}^{k} y_i(y_i-1)}{kl(l-1)} & \text{otherwise.} \end{cases} \tag{7}$$

where $y_i$ is the number of 1s in the $i$th row of the AT matrix.

For example, using Formula (6), the MMAC for the *AccountDialog* class is calculated as follows:

$$\mathrm{MMAC}(\mathit{AccountDialog}) = \frac{2(1) + 1(0) + 1(0) + 3(2)}{4(4)(3)} = 0.167$$

Using Formula (7), the AAC for the *AccountDialog* class is calculated as follows:

$$\mathrm{AAC}(\mathit{AccountDialog}) = \frac{4(3) + 1(0) + 1(0)}{4(4)(3)} = 0.25$$

The MMAC and AAC metrics are adaptations for the similarity definition introduced in [21] to measure the cohesion within a method in a class or a function in a procedural program.

### 4.2. AMC metric

The notion of similarity is applicable only when both elements considered are of the same entity. Therefore, the notion of similarity is applicable for method–method and attribute–attribute pairs, but it is not applicable for attribute–method pairs because attributes and methods are of two different types. In this case, the cohesion is the average number of attribute–method interactions represented in the AT matrix. In other words, the AMC is the ratio of the number of 1s in the AT matrix to the total size of the matrix. The AMC of a class C is defined formally as follows:

$$\mathrm{AAC(C)} = \begin{cases} 0 & \text{if } k = 0 \text{ or } l = 0 \\ \frac{\sum_{i=1}^{k} \sum_{j=1}^{l} m_{ij}}{kl} & \text{otherwise.} \end{cases} \tag{8}$$

Using Formula (8), AMC(*AccountDialog*) = 7/16 = 0.438.

### 4.3. MMIC metric

The AT matrix does not represent the cohesion between a pair of methods if one of the methods invokes the other when the invoked method does not have parameters of types that match the attributes. In this case, the number of 1s in the row of the invoked method in the AT matrix is zero. When the logical operator OR is applied to update the row of the caller using the algorithm given in Fig. 6, the row of the caller is not changed. In addition, the AT matrix does not represent the invoking relationship when the set of types of parameters used by the call recipient is a subset of the types of parameters used by the caller. In this case, when the algorithm in Fig. 6 is applied, the row of the caller is not changed. In order to consider the invocation relationship and assign it a weight when measuring the class cohesion, the MI matrix is involved. The notion of similarity is not represented in the MI matrix because the matrix explicitly represents method–method-invocation interactions. In this case, the cohesion is the average number of method–method-invocation interactions. This is represented by the ratio of the number of 1s in the MI matrix to the total size of the matrix. However, recursive method invocations should be excluded because we are interested in measuring the cohesion between pairs of different methods. As a result, the method–method invocation cohesion (MMIC) of a class C is formally defined as follows:

$$\mathrm{MMIC(C)} = \begin{cases} 0 & \text{if } k = 0, \\ 1 & \text{if } k = 1, \\ \frac{\sum_{i=1}^{k} \sum_{j=1, i \neq j}^{k} m_{ij}}{k(k-1)} & \text{otherwise.} \end{cases} \tag{9}$$

Using Formula (9), MMIC(*AccountDialog*) = 3/(4 * 3) = 0.25.

### 4.4. SCC metric

The SCC metric is defined as the weighted summation of the MMAC, AAC, AMC, and MMIC metrics. The SCC of a class C is defined for $k > 1$ and $l > 1$ as follows:

$$\mathrm{SCC(C)} = \frac{\mathrm{MP} * \mathrm{MMAC(C)} + \mathrm{AP} * \mathrm{AAC(C)} + \mathrm{MOP} * \mathrm{MMIC(C)} + lk * \mathrm{AMC(C)}}{\mathrm{MP} + \mathrm{AP} + \mathrm{MOP} + lk}, \tag{10}$$

where MP is the number of method pairs, AP is the number of distinct attribute-types pairs, and MOP is the number of method ordered pairs. By substituting MP, AP and MOP by their equivalencies in Formula (10) and considering all cases of $k$ and $l$ except when both are equal to 0, the SCC is formally defined as follows:

$$\mathrm{SCC}(C) = \begin{cases} 0 & \text{if } k = 0 \text{ and } l \geqslant 1, \\ 1 & \text{if } k = 1 \text{ and } l = 0, \\ \mathrm{MMIC}(C) & \text{if } k > 1 \text{ and } l = 0, \\ \frac{k(k-1)[\mathrm{MMAC}(C)+2\mathrm{MMIC}(C)]+l(l-1)\mathrm{AAC}(C)+2lk\mathrm{AMC}(C)}{3k(k-1)+l(l-1)+2lk} & \text{otherwise.} \end{cases} \tag{11}$$

Table 1 shows all of the possible scenarios for the values of the number of methods and attributes in a class and the intuition for their class cohesion. Comparing the results of applying the SCC metric for the cases given in Table 1 with the intuitive results shows that the SCC metric is consistent with intuition.

Using Formula (11), the SCC for the *AccountDialog* class is calculated as follows:

$$\mathrm{SCC}(\mathit{AccountDialog}) = \frac{4(3)[0.167 + 2(0.25)] + 4(3)(0.25) + 2(4)(4)(0.438)}{3(4)(3) + 4(3) + 2(4)(4)} = 0.313$$

In summary, SCC is based on information available during HLD phase in UML diagrams. The metric can easily account for class inheritance and direct and transitive interactions and it is based on more realistic assumptions than existing HLD cohesion metrics. The notions of method invocation and transitive interaction are not applicable for interfaces. To measure the interface cohesion using SCC, the MMIC metric must be left out and MMAC, AAC, and AMC have to be applied on the DAT matrix instead of AT. In this case, SCC is the weighted summation of the three applicable matrices: MMAC, AAC, and AMC. The following sections show that the SCC metric complies with the class cohesion necessary properties and has relatively high fault-prediction power compared to other proposed HLD and LLD cohesion metrics.

## 5. Theoretical validation

We validate SCC using the necessary properties for a class cohesion metric proposed by Briand et al. [20] and discussed in Section 2.1.

**Property SCC.1.** *The SCC metric satisfies the non-negativity and normalization property.*

**Proof.** The minimum value for the SCC metric for a class is 0 when the class has (1) no methods and one attribute; (2) one method whose parameter types do not match any of the attribute types; or (3) several methods whose parameter types do not match any of the attribute types and none of the methods invokes the other. The maximum value for the SCC metric for a class is 1 when the class has (1) one method and no attributes; (2) one method, one or more attributes, and each type of the attributes matches a method parameter type; (3) several methods, no attributes, and every method invokes another method directly or transitively; or (4) several methods, several attributes, each type of the attributes matches a parameter type for each method directly or transitively, and each method directly or transitively invokes every other method. As a result, the SCC metric ranges over the interval [0, 1], and it therefore satisfies the non-negativity and normalization property. □

**Property SCC.2.** *The SCC metric satisfies the null and maximum-values property.*

**Proof.** Given a class with the set of methods and attributes, if for every method, none of the parameter types match the attribute types and the method does not directly or transitively invoke another method (that is, the class has no cohesive interactions), the value of the SCC metric will be 0. On the other hand, if each type of attribute matches a parameter type for each method directly or transitively, and every method directly or transitively invokes every other method (that is, the class features all possible interactions), the value of SCC metric will be 1 (that is, the maximum possible value). Hence, the SCC metric satisfies the null- and maximum-values property. □

**Property SCC.3.** *The SCC metric satisfies the monotonicity property.*

**Proof.** Adding a cohesive interaction to the AT matrix is represented by changing an entry value from 0 to 1. Changing an entry from 0 to 1 increases the number of 1s in a column and a row. This increases the numerator value in Formula (8) (AMC metric). The value of the numerator in Formula (6) (MMAC metric) increases if the column in which the entry is changed features at least one other entry with value 1; otherwise, it remains the same. The value of the numerator in Formula (7) (AAC metric) increases if the row in which the entry is changed has at least one other entry with value 1; otherwise, it remains the same. Increasing the numerator in Formula (6) increases the value of the AMC metric because the denominator does not change unless the size of the matrix changes. The same applies to the MMAC and AAC metrics. Changing an entry in the AT matrix does not affect the MI matrix. As a result, adding a cohesive interaction represented in the AT matrix always increases the SCC value. □

Adding a cohesive interaction to the MI matrix is represented by changing the value of a non-diagonal entry from 0 to 1. This increases the numerator value in Formula (9) (MMIC metric), but it does not affect the denominator. In some cases, adding a cohesive interaction to the MI matrix causes a change in the values of certain entries in the AT matrix from 0 to 1, which, as discussed earlier, increases the value of the SCC metric. Therefore, adding a cohesive interaction represented in the MI matrix always increases

**Table 1**
The intuitive results for the class cohesion in different scenarios.

| $k$ | $l$ | Intuition |
|---|---|---|
| 0 | 0 | The class has no methods and no attributes, and therefore, the cohesion is not defined |
| 0 | >0 | The class has no methods, and therefore, its attributes are not related and the cohesion is the minimum |
| 1 | 0 | The class has one method, and therefore, it performs one cohesive task and its cohesion is the maximum |
| 1 | >0 | The class has one method and one or more attributes, and therefore, the cohesion depends on the attribute–attribute and attribute–method interactions |
| >1 | 0 | The class has several methods and no attributes, and therefore, the cohesion depends only on the method–method-invocation interactions |
| >1 | 1 | The class has several methods and one attribute, and therefore, the cohesion depends on the method–method invocation, method–method use of the attribute, and attribute–method interactions |
| >1 | >1 | The class has several methods and attributes, and therefore, the cohesion depends on the method–method invocation, method–method use of attributes, attribute–attribute, and attribute–method interactions |

the SCC metric value. As a result, adding a cohesive interaction represented in the AT or MI matrix always increases the SCC metric value, which means that the SCC metric satisfies the monotonicity property.

**Property MMAC.1 and Property AAC.1.** *The MMAC and AAC metrics satisfy the cohesive module property.*

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

$$\sum_{i=1}^{l} x_i(x_i-1) + \sum_{i=1}^{n} x_i(x_i-1) = \sum_{i=1}^{l+n} x_i(x_i-1) \text{ and } \sum_{i=1}^{k} y_i(y_i-1) + \sum_{i=1}^{m} y_i(y_i-1) = \sum_{i=1}^{k+m} y_i(y_i-1)$$

Suppose that MMAC(c1) $\geqslant$ MMAC(c2), then

$$\begin{aligned}
\frac{\sum_{i=1}^{l} x_i(x_i-1)}{lk(k-1)} &\geqslant \frac{\sum_{i=1}^{n} x_i(x_i-1)}{mn(m-1)} \Rightarrow mn(m-1)\sum_{i=1}^{l} x_i(x_i-1) \\
&\geqslant lk(k-1)\sum_{i=1}^{n} x_i(x_i-1) \\
&\Rightarrow [mn(m-1)+lk(k-1)]\sum_{i=1}^{l} x_i(x_i-1) \\
&\geqslant lk(k-1)[\sum_{i=1}^{n} x_i(x_i-1)+\sum_{i=1}^{l} x_i(x_i-1)] \Rightarrow \frac{\sum_{i=1}^{l} x_i(x_i-1)}{lk(k-1)} \\
&\geqslant \frac{\sum_{i=1}^{n} x_i(x_i-1)+\sum_{i=1}^{l} x_i(x_i-1)}{mn(m-1)+lk(k-1)} \\
&> \frac{\sum_{i=1}^{n} x_i(x_i-1)+\sum_{i=1}^{l} x_i(x_i-1)}{mn(m-1)+lk(k-1)+(l+n)km+(lm+nk)(k+m-1)} \\
&\Rightarrow \frac{\sum_{i=1}^{l} x_i(x_i-1)}{lk(k-1)} > \frac{\sum_{i=1}^{l+n} x_i(x_i-1)}{(l+n)(m+k)(k+m-1)} \rightarrow MMAC(c1) \\
&> MMAC(c3)
\end{aligned}$$

So, Max{MMAC(c1),MMAC(c2)}>MMAC(c3).

Similarly, the AAC of the split classes is greater than the AAC of the merged class. This means that both the MMAC and AAC metrics satisfy the cohesive-modules property. □

**Property AMC.1.** *The AMC metric satisfies the cohesive-modules property*

**Proof.** Merging the two unrelated classes c1 and c2 implies that none of the methods in each of the two split classes are shared and none of them share common attribute types. In terms of the AT matrix, this means that the total number of 1s in the AT matrix of the merged class equals the sum of the number of 1s in the AT matrices of both of the split classes, which means formally:

$$\sum_{i=1}^{k}\sum_{j=1}^{l} m_{ij} + \sum_{i=1}^{m}\sum_{j=1}^{n} m_{ij} = \sum_{i=1}^{k+m}\sum_{j=1}^{l+n} m_{ij}$$

Suppose that AMC(c1) $\geqslant$ AMC(c2), then

$$\begin{aligned}
\frac{\sum_{i=1}^{k}\sum_{j=1}^{l} m_{ij}}{kl} &\geqslant \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} m_{ij}}{mn} \Rightarrow mn\sum_{i=1}^{k}\sum_{j=1}^{l} m_{ij} \geqslant kl\sum_{i=1}^{m}\sum_{j=1}^{n} m_{ij} \\
&\Rightarrow (mn+kl)\sum_{i=1}^{k}\sum_{j=1}^{l} m_{ij} \\
&\geqslant kl\left(\sum_{i=1}^{m}\sum_{j=1}^{n} m_{ij} + \sum_{i=1}^{k}\sum_{j=1}^{l} m_{ij}\right) \Rightarrow \sum_{i=1}^{k}\sum_{j=1}^{l} m_{ij} kl \\
&\geqslant \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} m_{ij} + \sum_{i=1}^{k}\sum_{j=1}^{l} m_{ij}}{mn+kl} \\
&> \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} m_{ij} + \sum_{i=1}^{k}\sum_{j=1}^{l} m_{ij}}{mn+kl+kn+ml} \Rightarrow \frac{\sum_{i=1}^{k}\sum_{j=1}^{l} m_{ij}}{kl} \\
&> \frac{\sum_{i=1}^{k+m}\sum_{j=1}^{l+n} m_{ij}}{(k+m)(l+n)} \Rightarrow AMC(c1) > AMC(c3)
\end{aligned}$$

So, Max{AMC(c1), AMC(c2)} > AMC(c3), which means that the AMC metric satisfies the cohesive module property. □

**Property MMIC.1.** *The MMIC metric satisfies the cohesive module property*

**Proof.** Merging two unrelated classes c1 and c2 implies that none of the methods in each of the two split classes are shared and none of the methods in class c1 invoke methods in class c2 or vice versa. In terms of the MI matrix, this means that the total number of 1s in the MI matrix of the merged class equals the sum of the number of 1s in both of the split classes, which formally means:

$$\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} m_{ij} + \sum_{i=1}^{m}\sum_{j=1,i\neq j}^{m} m_{ij} = \sum_{i=1}^{k+m}\sum_{j=1,i\neq j}^{k+m} m_{ij}$$

Suppose that MMIC(c1) $\geqslant$ MMIC(c2), then

$$\begin{aligned}
\frac{\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} m_{ij}}{k(k-1)} &\geqslant \frac{\sum_{i=1}^{m}\sum_{j=1,i\neq j}^{m} m_{ij}}{m(m-1)} \Rightarrow m(m-1)\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} m_{ij} \\
&\geqslant k(k-1)\sum_{i=1}^{m}\sum_{j=1,i\neq j}^{m} m_{ij} \\
&\Rightarrow [k(k-1)+m(m-1)]\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} m_{ij} \\
&\geqslant k(k-1)\left(\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} m_{ij} + \sum_{i=1}^{m}\sum_{j=1,i\neq j}^{m} m_{ij}\right) \\
&\Rightarrow \frac{\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} m_{ij}}{k(k-1)} \\
&\geqslant \frac{\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} m_{ij} + \sum_{i=1}^{m}\sum_{j=1,i\neq j}^{m} m_{ij}}{k(k-1)+m(m-1)} \\
&> \frac{\sum_{i=1}^{k+m}\sum_{j=1,i\neq j}^{k+m} m_{ij}}{k(k-1)+m(m-1)+2mk} \Rightarrow \frac{\sum_{i=1}^{k}\sum_{j=1,i\neq j}^{k} m_{ij}}{k(k-1)} \\
&> \frac{\sum_{i=1}^{k+m}\sum_{j=1,i\neq j}^{k+m} m_{ij}}{(k+m)(k+m-1)} \Rightarrow \text{MMIC(c1)} > \text{MMIC(c3)}
\end{aligned}$$

So, Max{MMIC(c1),MMIC(c2)} > MMIC(c3), which means that the MMIC metric satisfies the cohesive-modules property. □

**Property SCC.4.** *The SCC metric satisfies the cohesive-modules property*

**Proof.** The SCC metric is the weighted sum of the MMAC, AAC, AMC, and MMIC metrics. Since the cohesion for the split classes is greater than the cohesion of the merged class for each of the four metrics, it is also greater for the SCC metric. Therefore, the SCC metric satisfies the cohesive module property. □

Showing that a cohesion metric has the expected mathematical properties increases the chances for the metric to be a meaningful quality indicator. However, it does not necessarily guarantee it. The following section reports on large-scale empirical investigations that demonstrate the validity of the assumptions underlying SCC and its superior fault prediction capability when compared to other HLD and LLD cohesion metrics.

## 6. Empirical validation

We present three analyses. The first explores the correlations among 10 cohesion metrics, including SCC and well known other metrics, and applies principal component analysis [39] to explore the orthogonal dimensions within this set of cohesion metrics. The goal is to confirm that SCC is indeed contributing new information. The second and third analyses explore the extent to which the 10 class cohesion metrics can explain the presence of faults in classes. Metrics are first considered individually, and then as combinations of predictors. If SCC has indeed better properties and relies on more realistic assumptions, then we would expect it to be a better quality indicator and, for example, more accurately predict faults. This is a common widely used and accepted assumption in many studies (e.g., [20,40–43]). Note that the goal here is *not* to build prediction models as many other factors besides cohesion would have to be accounted for. The goal is rather to determine if there is empirical evidence, whether direct or indirect, that SCC is indeed a well-defined measure, complementary or better than existing, comparable cohesion metrics. The fact that SCC explains more of the variation in fault occurrences is one piece of empirical evidence suggesting that this metric indeed capture cohesion better.

### 6.1. Software systems and metrics

We chose four Java open source software systems from different domains: Art of Illusion v.2.5 [44], GanttProject v.2.0.5 [45], JabRef v.2.3 beta 2 [46], and Openbravo v.0.0.24 [47]. 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 number of classes, (3) are from different domains, and (4) have available source code and fault repositories.

We selected three HLD and six LLD cohesion metrics to compare with SCC. The three HLD metrics, CAMC, NHD, and SNHD, had not been previously studied empirically. Though these metrics were originally applied to C++ classes, their concepts are general, and therefore, there is no reason not to apply them to Java classes[1]. These metrics were selected because they measure cohesion at the same design level as SCC, and therefore, they were used to compare the SCC to existing HLD metrics in terms of fault-prediction power. The six LLD metrics, LCOM1, LCOM2, LCOM3, Coh, TCC, and LCC, were selected because they had been extensively studied and compared to each other [5,40,43,48,49], and therefore, our results can be compared to those obtained in previous empirical studies. In addition, these metrics were selected to compare the fault-prediction power of the HLD and LLD metrics.

We applied the considered metrics to 1337 non-trivial, selected classes among 2035 classes from the four open-source systems. We excluded all classes for which at least one of the metrics was undefined. For example, classes consisting of single methods were excluded because their CAMC, NHD, SNHD, TCC, and LCC values are undefined. In addition, classes not consisting of any attributes were excluded because their NHD, SNHD, TCC, and LCC values are undefined. Classes in which all methods do not have parameters are excluded because their NHD values are undefined. An advantage of the SCC metric is that it is defined in all cases, as discussed in Section 4.4. Therefore, none of the classes were excluded because of an undefined SCC value. Excluding the classes that have undefined cohesion values using some of the considered metrics 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 are undefined in this case. We developed our own Java tool to automate the cohesion measurement process for Java classes using the 10 considered metrics including SCC. Though we focus on design metrics in this paper, that is metrics that can be measured on design models such as UML diagrams, in our experiment we collected the cohesion data through source code analysis as no UML design diagrams were available for the four open-source systems considered. Note, however, that only design information available during HLD and LLD was extracted, as discussed in Section 2.4. Our analysis tool analyzed the Java source code, extracted the required information to build the matrices, and calculated the cohesion values using the 10 considered metrics. Table 2 shows descriptive statistics for each cohesion measure including the minimum, 25% quartile, mean, median, 75% quartile, maximum value, and standard deviation. As indicated in [40], LCOM-based metrics feature extreme outliers due to accessor methods that typically reference single attributes. The 25% quartile, mean, median, and 75% quartile for SCC indicate that the considered classes have relatively few method invocations and low degrees of similarity among methods and attributes.

### 6.2. Investigating assumptions

The method–attribute interactions are not precisely known during the HLD phase because of the absence of method implementations or precise postconditions. As a result, all HLD metrics must rely on assumptions, the actuality of which needs to be investigated. In this paper, the SCC metric assumes that the set of attribute types accessed by a method is the intersection of the set of

**Table 2**
Descriptive statistics for the cohesion measures.

| Metric | Min | 25% | Mean | Med | 75% | Max | Std. Dev. |
|---|---|---|---|---|---|---|---|
| SCC | 0.00 | 0.04 | 0.18 | 0.08 | 0.18 | 1.00 | 0.26 |
| CAMC | 0.03 | 0.14 | 0.27 | 0.23 | 0.33 | 1.00 | 0.17 |
| NHD | 0.00 | 0.42 | 0.56 | 0.61 | 0.75 | 1.00 | 0.26 |
| SNHD | −1.00 | −0.36 | −0.13 | 0.00 | 0.00 | 1.00 | 0.46 |
| LCOM1 | 0.00 | 2.00 | 58.37 | 9.00 | 36.00 | 3401 | 201.89 |
| LCOM2 | 0.00 | 0.00 | 39.22 | 2.00 | 22.00 | 2886 | 160.30 |
| LCOM3 | 0.00 | 1.00 | 1.20 | 1.00 | 1.00 | 8.00 | 0.76 |
| Coh | 0.00 | 0.22 | 0.44 | 0.38 | 0.62 | 1.00 | 0.29 |
| TCC | 0.00 | 0.24 | 0.52 | 0.50 | 0.80 | 1.00 | 0.34 |
| LCC | 0.00 | 0.33 | 0.64 | 0.72 | 1.00 | 1.00 | 0.36 |

[1] The key differences between C++ and java, such as multiple inheritance and the way the destructor is invoked, do not play a role in building the parameter-occurrence matrix.

this method's parameter types and the set of class attribute types. This assumption is captured by the DAT matrix that shows the matching between the attribute types and the method parameter types, as illustrated in Section 3.3. On the other hand, the parameter-occurrence matrix [7] is based on the assumption that the set of attribute types accessed by a method is the intersection of this method's parameter types and the set of parameter types of all methods in the class. The matrix shows, for each method, the matching between the types of the method parameters and the types of parameters for all methods in the class. We empirically studied the correctness of our assumption and that of Counsell using all the considered classes in our study. For each class, we calculated the percentage of cells in the DAT and parameter-occurrence matrices matching the results of source code analysis, which is able to precisely identify the access of each attribute in each method. For each of the four considered systems, we calculated the average of the matching percentages of all considered classes in the system and reported the results in Table 3. In addition, for all classes considered in this empirical study, we investigated the occurrences of the two problems, stated in Section 3.3, in using the parameter-occurrence matrix. For the first problem, we counted for each class the number of parameter types, in the occurrence-matrix, that do not match the types of the attributes. We calculated the percentage represented by this number among all parameter types included in the parameter-occurrence matrix. In the fourth column of Table 3, we reported the average of the percentages of the parameter types that do not match attribute types in all considered classes in each system. Similarly, for the second problem, we counted the number of attribute types not included in the parameter-occurrence matrix. We calculated the percentage represented by this number among all distinct attribute types in the class. Finally, in the last column of Table 3, we reported the average of the percentages of the attribute types not included in the parameter-occurrence matrix in all considered classes in each system. The last row in Table 3 shows the corresponding percentages over all considered classes in the considered systems.

The results show that the average matching percentage of the DAT matrix (65.49%) is much better than the average matching percentage of the parameter-occurrence matrix (20.87%). In other words, this means that, on average, 65.49% of the attributes accessed by the methods were detected using our assumption, whereas, on average, 20.87% of the attributes accessed by the methods were detected using the assumption of Counsell. This suggests that our metric takes more interactions into account than other existing HLD metrics because it is based on the DAT matrix which appears to predict method–attribute interactions better than the parameter-occurrence matrix. The results also show that the average percentage of parameter types that do not match attribute types (57.51%) and the average percentage of the attribute types that are not included in the parameter-occurrence matrix (45.61%) are relatively high. In other words, this means that, on average, 57.51% of the types included in the parameter-occurrence matrix are unnecessary in representing the use of attributes in the methods of a class. Furthermore, on average, 45.61% of the attribute types are not represented in the parameter-occurrence matrix. This indicates that most often the parameter-occurrence matrix is unrealistic in representing the use of attributes in the methods of a class. In most cases, either the types of the attributes are not represented in the parameter-occurrence matrix or the types of the parameters do not match any of the types of the attributes. As a result, we cannot rely on the parameter-occurrence matrix as an indicator for the accessibility of the attributes by the methods. Although we showed how to include inherited methods and attributes when measuring SCC, the following analyses do not consider inheritance. Empirically studying the effect of including or excluding inherited attributes and methods is left open for further research.

**Table 3**
The matching percentages of the DAT and parameter-occurrence matrices.

| Systems | Matching percentage of DAT matrix | Matching percentage of parameter-occurrence matrix | Percentage of parameter types not matching attribute types | Percentage of attribute types not included in parameter-occurrence matrix |
|---|---|---|---|---|
| Art of Illusion | 65.12 | 20.08 | 59.93 | 42.04 |
| GanttProject | 68.31 | 23.29 | 56.02 | 43.03 |
| JabRef | 65.39 | 22.74 | 51.77 | 44.16 |
| Openbravo | 63.10 | 16.97 | 63.25 | 54.70 |
| **Overall classes** | **65.49** | **20.87** | **57.51** | **45.61** |

### 6.3. Correlation and principal component analyses

Principal Component Analysis (PCA) [39] is a technique used here to identify and understand the underlying orthogonal dimensions that explain the relations between the cohesion metrics [43]. In addition, it is useful to demonstrate that the proposed metric captures new measurement dimensions. For each pair of considered cohesion metrics, we used Mahalanobis Distance [50] to detect outliers and we found that removing the outliers does not make significant differences in the final PCA results. We calculated the nonparametric Spearman correlation coefficient [51] among the considered cohesion metrics. Table 4[2] shows the resulting correlations among the considered metrics accounting for all four systems. They are all statistically significant ($p$-value <0.0001). Most cohesion metrics are weakly or moderately intercorrelated, though much stronger correlations are observed for TCC and LCC (0.90), LCOM1 and LCOM2 (0.8), and CAMC, NHD, and LCOM1, respectively. SCC is also at best moderately correlated with other cohesion metrics, the maximum being CACM with a Spearman correlation of 0.51. This is to be expected as CACM is the metric that is most similar to SCC.

To obtain the principal components (PCs), we used the *varimax* rotation technique [52,53] 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 six PCs that capture 93.44% 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:* CAMC, NHD, LCOM1, LCOM2, Coh, TCC, and LCC. These metrics consider the share or use of attributes or their representatives (i.e., parameter types in CAMC and NHD) in the methods of a class as the basis for measuring cohesion. In addition, these metrics have zero lower bounds.

*PC2:* LCOM1, LCOM2, TCC, and LCC. These metrics consider the share of attributes in the methods of a class as the basis for measuring cohesion.

*PC3:* SCC and CAMC. These are HLD metrics considering the parameter types of the methods of the classes.

*PC4:* SNHD. This metric only indirectly captures cohesion. Originally, SNHD was introduced to represent the closeness of the NHD

[2] Note that we use the absolute value of SNHD, where values close to 0 indicate high cohesion and values close to 1 indicate low cohesion.

**Table 4**
Spearman rank correlations among the cohesion metrics.

| Metric | CAMC | NHD | \|SNHD\| | LCOM1 | LCOM2 | LCOM3 | Coh | TCC | LCC |
|---|---|---|---|---|---|---|---|---|---|
| SCC | 0.51 | −0.44 | −0.16 | −0.45 | −0.33 | −0.11 | 0.41 | 0.31 | 0.27 |
| CAMC | 1.00 | −0.88 | −0.31 | −0.79 | −0.54 | −0.30 | 0.52 | 0.31 | 0.18 |
| NHD | | 1.00 | 0.43 | 0.84 | 0.59 | 0.30 | −0.58 | −0.35 | −0.21 |
| \|SNHD\| | | | 1.00 | 0.56 | 0.40 | 0.18 | −0.38 | −0.20 | −0.14 |
| LCOM1 | | | | 1.00 | 0.80 | 0.34 | −0.75 | −0.57 | −0.42 |
| LCOM2 | | | | | 1.00 | 0.34 | −0.77 | −0.73 | −0.61 |
| LCOM3 | | | | | | 1.00 | −0.19 | −0.12 | −0.08 |
| Coh | | | | | | | 1.00 | 0.733 | 0.58 |
| TCC | | | | | | | | 1.00 | 0.90 |
| LCC | | | | | | | | | 1.00 |

**Table 5**
Loading matrix.

| | PC1 | PC2 | PC3 | PC4 | PC5 | PC6 |
|---|---|---|---|---|---|---|
| Eigenvalue | 3.89 | 1.88 | 1.36 | 0.87 | 0.72 | 0.63 |
| Percent | 38.89 | 18.77 | 13.60 | 8.69 | 7.21 | 6.28 |
| Cum. Per. | 38.89 | 57.66 | 71.26 | 79.95 | 87.16 | 93.44 |
| SCC | 0.48 | 0.06 | **0.50** | 0.43 | 0.04 | **0.56** |
| CAMC | **0.68** | −0.06 | **0.52** | 0.14 | 0.05 | −0.28 |
| NHD | **−0.75** | −0.02 | −0.40 | 0.09 | −0.03 | 0.36 |
| \|SNHD\| | −0.45 | 0.05 | −0.24 | **0.79** | 0.13 | −0.30 |
| LCOM1 | **−0.60** | **0.68** | 0.30 | <0.01 | −0.25 | −0.04 |
| LCOM2 | **−0.59** | **0.67** | 0.37 | −0.03 | −0.21 | −0.06 |
| LCOM3 | −0.47 | 0.38 | 0.15 | −0.17 | **0.76** | 0.03 |
| Coh | **0.82** | 0.34 | −0.01 | −0.02 | 0.09 | −0.03 |
| TCC | **0.73** | **0.54** | −0.36 | 0.03 | 0.03 | <0.01 |
| LCC | **0.53** | **0.62** | −0.49 | 0.03 | −0.05 | 0.06 |

metric to the maximum value of NHD compared to the minimum value.

*PC5:* LCOM3. This metric is based on counting the number of connected components in the class representative graph instead of counting the number of shared or used attributes.

*PC6:* SCC. This is our new HLD metric that considers method–method, method–attribute, and attribute–attribute relationships. In addition, it considers method invocations and transitive use of attributes, and is more refined than the others in terms of measuring the degree of similarity among the attributes or the methods.

The PCA results show that SCC metric captures a measurement dimension of its own as it is the only significant factor in PC6, though it also contributes to PC3. This supports the fact that SCC captures cohesion aspects that are not addressed by any of the cohesion metrics considered in this analysis, thus confirming the results of correlation analysis.

### 6.4. Predicting faults in classes

To study the relationship between the values of collected metrics and the extent to which a class is fault-prone, we applied logistic regression [54], a standard and mature statistical method based on maximum likelihood estimation. This method is widely applied to predict fault-prone classes (i.e., [20,40,41,43]) and though other analysis methods, such as the methods discussed by Briand and Wuest [55] and Arisholm et al. [56], 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 of each metric separately, whereas multivariate regression is applied to study the fault prediction of different combinations of metrics to determine whether SCC improves the fit of these combinations.

We collected fault data for the classes in the considered software systems from publicly available fault repositories. 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. 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., [20,40,41,43]) and is necessary to enable statistical analysis.

Univariate regression results are reported in Table 6. Estimated regression coefficients are reported, as well as their 95% confidence

**Table 6**
Univariate logistic regression results.

| Design phase | Metric | Std. Coeff. | Odd ratio | Std. Error | 95% Confidence Interval Coeff. | 95% Confidence interval odd ratio | *p*-Value | Precision | Recall | ROC area |
|---|---|---|---|---|---|---|---|---|---|---|
| HLD | SCC | −0.68 | 0.51 | 0.07 | [−0.81,−0.55] | [0.44,0.58] | <0.0001 | 69.6 | 68.8 | 69.0 |
| | CAMC | −0.47 | 0.62 | 0.06 | [−0.59,−0.35] | [0.56,0.70] | <0.0001 | 61.9 | 64.8 | 63.1 |
| | NHD | 0.32 | 1.38 | 0.06 | [0.21,0.43] | [1.23,1.54] | <0.0001 | 60.0 | 64.0 | 59.4 |
| | \|SNHD\| | 0.18 | 1.20 | 0.07 | [0.04,0.32] | [1.04,1.38] | 0.002 | 41.3 | 64.2 | 53.1 |
| LLD | LCOM1 | 1.06 | 2.89 | 0.20 | [0.66,1.46] | [1.94,4.30] | <0.0001 | 41.3 | 64.2 | 62.3 |
| | LCOM2 | 1.22 | 3.39 | 0.25 | [0.74,1.70] | [2.10,5.47] | <0.0001 | 41.3 | 64.2 | 62.7 |
| | LCOM3 | 0.13 | 1.14 | 0.06 | [0.004,0.26] | [1.00,1.29] | 0.049 | 41.3 | 64.2 | 50.3 |
| | Coh | −0.53 | 0.59 | 0.06 | [−0.65,−0.42] | [0.52,0.66] | <0.0001 | 63.2 | 65.8 | 65.2 |
| | TCC | −0.41 | 0.67 | 0.059 | [−0.52,−0.29] | [0.60,0.75] | <0.0001 | 57.2 | 63.7 | 61.1 |
| | LCC | −0.33 | 0.72 | 0.060 | [−0.45,−0.21] | [0.64,0.81] | <0.0001 | 41.3 | 64.2 | 59.2 |

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 2. 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 6 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* [54] 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 6. As an example, for SCC, the probability of fault occurrence when there is an increase of one standard deviation in SCC is estimated to decrease by 49%. Those can be easily compared across cohesion metrics. We use a typical significance threshold ($\alpha = 0.05$) 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: $\alpha/10 = 0.005$ [57].

To evaluate the prediction accuracy of logistic regression models, we used the traditional precision and recall evaluation criteria [58]. 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 the 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 the faulty classes correctly or incorrectly classified as faulty. Such criteria, however, require the selection of a probability threshold to predict classes as faulty or not. Following the recommendation in [48], a class is classified as faulty if its predicted probability of containing a fault is higher than a threshold selected such that the percentage of classes that are classified as faulty is roughly the same as the percentage of classes that actually are faulty.

To evaluate the performance of a prediction model regardless of any particular threshold, we used the receiver operating characteristic (ROC) curve [59]. In the context of fault-prediction problems, 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 depicts the ability of the model to correctly rank classes as faulty or non-faulty. The ROC area of 100% represents a perfect model that classifies all classes correctly. The larger the ROC area, the better the model is in 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 ability of the metrics, we used cross-validation, a procedure in which the data set is partitioned into *k* subsamples. The regression model is then built and evaluated *k* times. Each time, a different subsample is used to evaluate the precision, recall, and ROC area of the model, and the remaining subsamples are used as training data to build the regression model.

The results in Table 6 lead to the following conclusions:

1. Except for LCOM3, all considered cohesion metrics are statistically significant at $\alpha = 0.005$ (i.e., their coefficients are significantly different from 0).
2. SCC is the best metric among the 10 considered HLD and LLD metrics, in terms of precision, recall, and ROC area.
3. SCC has a significantly better precision than the other HLD and LLD metrics, especially when compared to SNHD and all LLD metrics but Coh.
4. SCC has slightly better recall than the other HLD and LLD metrics.
5. SCC has higher ROC areas than the other HLD and LLD metrics, especially when compared to SNHD and LCOM3.
6. Among the considered HLD metrics, SNHD is the worst metric in terms of precision and ROC area and NHD is the worst metric in terms of recall. In addition, among the considered LLD metrics, LCOM3 is the worst metric in terms of precision, recall, and ROC area.
7. As expected, the estimated regression coefficients for the inverse cohesion measures LCOM1, LCOM2, LCOM3, NHD, and SNHD are positive, whereas those for the straight cohesion measures SCC, Coh, TCC, LCC, and CAMC are negative. In each case, this indicates an increase in the predicted probability of fault detection as the cohesion of the class decreases.
8. SCC has the largest standardized coefficient among all considered HLD cohesion metrics. This is confirmed by a smaller odd ratio (0.51), thus suggesting that an increase in SCC has a stronger impact on reducing fault occurrence probability. Once again, as expected, CACM has the next smaller odd ratio (0.62) while those of the other HLD metrics are much higher. But even for CACM the 95% odd ratio confidence intervals are barely overlapping. To compare the odd ratios of inverse cohesion metrics—which have coefficients above one—with SCC, 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 $1/1.22 = 0.82$.
9. SCC is one of the metrics with the smallest p-value (which can show statistical significance of the regression coefficient) among all considered HLD and LLD cohesion metrics.

Let us now turn our attention to multivariate analysis and the role of SCC in building class fault-proneness prediction models based on cohesion metrics. To study whether an optimal yet minimal multivariate model would contain SCC, we used a backward selection process where all HLD 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 7 show that both SCC and CAMC remain in the prediction model as significant covariates. This somehow shows that they are complementary in predicting faults. Note that the resulting model is the first model represented in Table 8. Interpreting regression coefficients in multivariate models with interacting covariates is always a difficult exercise.

Another way to look at the impact of SCC in multivariate models is to assess whether models containing SCC perform better than their counterparts. Here regression analysis was applied only to combinations of the four HLD metrics considered in this paper:

**Table 7**
The model based on HLD metrics.

| Metric | Std. Coeff. | Std. Error | *p*-Value |
|---|---|---|---|
| SCC | −0.58 | 0.28 | <0.001 |
| CAMC | −0.26 | 0.41 | 0.001 |

**Table 8**
Multivariate logistic regression results.

| Model | Metrics | Precision | Recall | ROC area |
|---|---|---|---|---|
| A | 1 + 2 | 68.4 | 68.6 | 68.5 |
| B | 1 + 3 | 69.1 | 68.8 | 68.0 |
| C | 1 + 4 | 69.6 | 69.0 | 68.1 |
| D | 2 + 3 | 60.2 | 63.9 | 63.0 |
| E | 2 + 4 | 62.0 | 65.0 | 62.8 |
| F | 3 + 4 | 60.0 | 64.0 | 58.8 |
| G | 1 + 2 + 3 | 68.3 | 68.5 | 68.6 |
| H | 1 + 2 + 4 | 68.9 | 68.9 | 68.2 |
| I | 1 + 3 + 4 | 69.3 | 69.0 | 67.5 |
| J | 2 + 3 + 4 | 60.2 | 63.9 | 62.7 |
| K | 1 + 2 + 3 + 4 | 68.1 | 68.4 | 68.2 |

1:SCC, 2:CAMC, 3:NHD, 4:|SNHD|

SCC, CAMC, NHD, and the absolute value of SNHD. We chose these four metrics because our goal is to compare SCC with other HLD measures. By combining them into multivariate models we want to determine whether SCC improves the fit of these models and is therefore complementary or a more optimal option than other metrics to explain class fault-proneness. Table 8 reports the results of several multivariate logistic regression models. The first column of Table 8 shows the model identifier, and the second column shows the identifiers of the metrics that are combined in the model. The rest of the columns show the resulting evaluations for each of the possible combinations.

The results of the multivariate logistic regression show that the best models are those that combine SCC with other metrics (i.e., models A, B, C, G, H, I, and K shown in Table 8). Furthermore, the precision, recall, and ROC area results for these models are almost the same regardless of how the other metrics are combined with SCC. These results are also close to those obtained when SCC is used on its own (see Table 6). Further, significant differences are observed when SCC is removed from a model (e.g., comparing 1 + 2 + 3 and 2 + 3). SCC therefore, seems to be the main driver, among cohesion measures, of the presence of faults in classes, though SCC is partly correlated to other metrics.

As a result, the empirical results above show that the SCC metric predicts faulty classes more accurately than the other nine selected cohesion metrics considered individually or in combination. This is particularly true when compared to the other three HLD cohesion metrics.

### 6.5. Threats to validity

#### 6.5.1. Construct validity

The SCC metric makes an important assumption: an attribute is likely to be accessed within a method if that method has a parameter of a type that matches an attribute type. One of the key limitations of using attribute and parameter types is that two or more attributes may have the same type. In this case, the metric coalesces such attributes into a single attribute. An implication of this approach is that it is difficult to tell which attribute is expected to be accessed by a method when the method has a type that matches a type that is associated with several attributes. Another implication is that two methods can unintentionally be considered cohesive because their parameters share the type of several attributes while in fact one of the methods accesses a certain attribute and the other accesses a different attribute. Another limitation of our metric is the assumption that two attributes are related if their types match the parameter types of a method, whereas the two attributes might be unrelated within the method. Though potential problems regarding this assumption must be carefully considered, our empirical results show that our basic assumption is met most of the time as reported in Section 6.2.

#### 6.5.2. 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., [60–62]), 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.

#### 6.5.3. 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 [56]. However, our goal here is not to predict faults in classes, but to investigate whether there is empirical evidence that SCC 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 HLD 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 6.4.

## 7. Conclusions and future work

This paper introduces a new cohesion metric (SCC) that addresses a number of problems arising from existing metrics. It is defined to be usable during the High-Level Design (HLD) of object-oriented software, which allows early designs to be assessed in terms of cohesion, in a way that is firmly grounded in theory and strongly supported by empirical evidence. SCC is based on more realistic, empirically verified assumptions than other comparable metrics. From a theoretical standpoint, it also accounts for all types of interactions between class members: method–method interactions caused by the sharing of attribute types; attribute-attribute interactions caused by the expected use of attributes within the methods; attribute–method interactions; and method–method-invocation interactions. Both direct and transitive interactions are considered. The metric uses two types of matrices that can be constructed using class and communication diagrams, commonly used in high-level design with the Unified Modeling Language (UML). The metric satisfies mathematical properties that are expected for cohesion measures and have been widely used in the literature. A large-scale empirical study based on four open-source systems is also reported showing how SCC and 10 other well-known cohesion metrics relate to each other and the occurrence of faults. Results show that SCC captures a cohesion measurement dimension of its own and was the metric that most strongly related to fault occurrences in statistical terms. As with any design metric, SCC has several limitations as it targets HLD and is therefore based on assumptions that are necessary as source code information is not available. However, the aim of HLD metrics

is not to measure class cohesion with maximum precision, but rather to approximate it to enable early assessments and decision-making. Empirical results show that our assumptions are met most of the time and that our approximation is good enough for the metric to exhibit a strong relationship with fault occurrences.

SCC can be improved in several ways, such as effectively solving the problem of having several attributes of the same type. Our metric does not distinguish between attributes and methods of different accessibility levels (i.e., public, private, and protected). Studying the effect of considering or ignoring private and protected attributes and methods on the computation of SCC and its fault-prediction power are left open for future research. Assessing empirically the impact of inheritance on SCC and its fault-prediction power is also relevant but requires complex automation. Future work will investigate whether complementary, relevant cohesion information can be provided by UML diagrams other than class and communication diagrams. In addition, in the future, we plan to introduce a similar LLD class cohesion metric and to study it empirically.

## Acknowledgments

The authors would like to acknowledge the support of this work by Kuwait University Research Grant WI03/07. In addition, the authors would like to thank Professor Steve Counsell for his insightful comments that helped improve the paper, Walid Bahsow for developing the class cohesion measuring tool and Manal Al-Khousi, Saqiba Sulman, Sharefa Al-Kandari, Methayell Al-Mitrik, Maryam Al-Hasawi, and Amina Bin Ali for collecting the cohesion results.

## References

[1] N. Fenton, S. Pfleeger, Software metrics: a rigorous and practical approach, course technology, second ed., 1998.
[2] J. Bieman, L. Ott, Measuring functional cohesion, IEEE Transactions on Software Engineering 20 (8) (1994) 644–657.
[3] L. Briand, C. Bunse, J. Daly, A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs, IEEE Transactions on Software Engineering 27 (6) (2001) 513–530.
[4] Z. Chen, Y. Zhou, B. Xu, A novel approach to measuring class cohesion based on dependence analysis, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 377–384.
[5] L.C. Briand, S. Morasca, V.R. Basili, Defining and validating measures for object-based high-level design, IEEE Transactions on Software Engineering 25 (5) (1999) 722–743.
[6] J. Bansiya, L. Etzkorn, C. Davis, W. Li, A class cohesion metric for object-oriented designs, Journal of Object-Oriented Program 11 (8) (1999) 47–52.
[7] S. Counsell, S. Swift, 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) 123–149.
[8] S.R. Chidamber, 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) 197–211.
[9] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[10] M. Hitz, B. Montazeri, Measuring coupling and cohesion in object oriented systems, in: Proceedings of the International Symposium on Applied Corporate Computing, 1995, pp. 25–27.
[11] J.M. Bieman, B. Kang, Cohesion and reuse in an object-oriented system, in: Proceedings of the 1995 Symposium on Software reusability, Seattle, Washington, United States, 1995, pp. 259–262.
[12] L. Badri, M. Badri, A proposal of a new class cohesion criterion: an empirical study, Journal of Object Technology 3 (4) (2004) 145–159.
[13] J. Wang, Y. Zhou, L. Wen, Y. Chen, H. Lu, B. Xu, D. MC, A more precise cohesion measure for classes, Information and Software Technology 47 (3) (2005) 167–180.
[14] L. Fernández, R. Peña, A sensitive metric of class cohesion, International Journal of Information Theories and Applications 13 (1) (2006) 82–91.
[15] S. Counsell, E. Mendes, S. Swift, Comprehension of object-oriented software cohesion: the empirical quagmire, in: Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002), 2002, pp. 33–42.
[16] H. Gomaa, Designing concurrent, distributed, and real-time applications with UML, Addison-Wesley Object Technology Series, Addison-Wesley Professional, 2000.
[17] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, second ed., Prentice Hall PTR, 2001.
[18] B. Bruegge, A. Dutoit, Object Oriented Software Engineering, third revised ed., Prentice Hall, 2009.
[19] B. Kitchenham, S.L. Pfleeger, N. Fenton, Towards a framework for software measurement validation, IEEE Transactions on Software Engineering 21 (12) (1995) 929–944.
[20] L.C. Briand, J. Daly, J. Wuest, A unified framework for cohesion measurement in object-oriented systems, Empirical Software Engineering – An International Journal 3 (1) (1998) 65–117.
[21] J. Al Dallal, Software similarity-based functional cohesion metric, IET Software 3 (1) (2009) 46–57.
[22] M. Choi, J. Lee, J. Ha, A component cohesion metric applying the properties of linear increment by dynamic dependency relationships between classes, in: Proceedings of the Computational Science and Its Applications – ICCSA 2006.
[23] J. Al Dallal, Mathematical validation of object-oriented class cohesion metrics, International Journal of Computers 4 (2) (2010) 45–52.
[24] M. Genero, M. Piattini, C. Caleron, A survey of metrics for UML class diagrams, Journal of Object Technology 4 (9) (2005) 59–92.
[25] E. Yourdon, L. Constantine, Structured Design, Prentice-Hall, Englewood Cliffs, 1979.
[26] T. Emerson, A discriminant metrics for module cohesion, in: Proceedings of the 7th International Conference on Software Engineering, 1984, pp. 294–303.
[27] A. Lakhotia, Rule-based approach to computing module cohesion, in: Proceedings of the 15th international conference on Software Engineering, Baltimore, US, 1993, pp. 35–44.
[28] L. Ott, J. Thuss, Slice based metrics for estimating cohesion, in: Proceedings of the First International Software Metrics Symposium, Baltimore, 1993, pp. 71–81.
[29] T. Meyers, D. Binkley, An empirical study of slice-based cohesion and coupling metrics, ACM Transactions on Software Engineering Methodology 17 (1) (2007) 2–27.
[30] J. Al Dallal, Efficient program slicing algorithms for measuring functional cohesion and parallelism, International Journal of Information Technology 4 (2) (2007) 93–100.
[31] D. Troy, S. Zweben, Measuring the quality of structured designs, Journal of Systems and Software 2 (1981) 113–120.
[32] J. Bieman, B. Kang, Measuring design-level cohesion, IEEE Transactions on Software Engineering 24 (2) (1998) 111–124.
[33] W. Li, S.M. Henry, Maintenance metrics for the object oriented paradigm, in: Proceedings of 1st International Software Metrics Symposium, Baltimore, MD, 1993, pp. 52–60.
[34] B. Henderson-Sellers, Object-Oriented Metrics Measures of Complexity, Prentice-Hall, 1996.
[35] J. Al Dallal, L. Briand, A Precise method–method interaction-based cohesion metric for object-oriented classes, ACM Transactions on Software Engineering and Methodology (TOSEM) (2010), in press.
[36] J. Al Dallal, A design-based cohesion metric for object-oriented classes, in: Proceedings of the International Conference on Computer and Information Science and Engineering (CISE 2007), Venice, Italy, November 2007.
[37] D. Pilone, N. Pitman, UML 2.0 in a Nutshell, second ed., O'Reilly Media, Inc., 2005. pp. 234.
[38] Object Management Group (OMG) – UML, UML resource page, http://www.uml.org/, July 2010.
[39] G. Dunteman, Principal components analysis, Saga University Paper No. 7–69, Saga Publications, US, 1989, pp. 96.
[40] L.C. Briand, J. Wüst, H. Lounis, Replicated case studies for investigating quality factors in object-oriented designs, Empirical Software Engineering 6 (1) (2001) 11–58.
[41] T. Gyimothy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Transactions on Software Engineering 3 (10) (2005) 897–910.
[42] K. Aggarwal, Y. Singh, A. Kaur, R. Malhotra, Investigating effect of design metrics on fault proneness in object-oriented systems, Journal of Object Technology 6 (10) (2007) 127–141.
[43] A. Marcus, D. Poshyvanyk, R. Ferenc, Using the conceptual cohesion of classes for fault prediction in object-oriented systems, IEEE Transactions on Software Engineering 34 (2) (2008) 287–300.
[44] Illusion. Available from: <http://sourceforge.net/projects/aoi/> February 2009.
[45] GanttProject. Available from: <http://sourceforge.net/projects/ganttproject/> February 2009.
[46] JabRef. Available from: <http://sourceforge.net/projects/jabref/> February 2009.
[47] Openbravo. Available from: <http://sourceforge.net/projects/openbravopos> February 2009.
[48] L.C. Briand, J. Wust, J. Daly, V. Porter, Exploring the relationship between design measures and software quality in object-oriented systems, Journal of System and Software 51 (3) (2000) 245–273.
[49] J. Al Dallal, Measuring the discriminative power of object-oriented class cohesion metrics, IEEE Transactions on Software Engineering (2010), in press.
[50] V. Barnett, T. Lewis, Outliers in Statistical Data, third ed., John Wiley and Sons, 1994.
[51] S. Siegel, J. Castellan, Nonparametric Statistics for the Behavioral Sciences, second ed., McGraw-Hill, 1988.

[52] I.T. Jolliffe, Principal component analysis, Springer, 1986.
[53] G. Snedecor, W. Cochran, Statistical Methods, eight ed., Blackwell Publishing Limited, 1989.
[54] D. Hosmer, S. Lemeshow, Applied Logistic Regression, second ed., Wiley Interscience, 2000.
[55] L.C. Briand, J. Wust, Empirical studies of quality models in object-oriented systems, Advances in Computers, Academic Press, 2002. pp. 97–166.
[56] E. Arisholm, L.C. Briand, 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.
[57] H. Abdi, Bonferroni and Sidak corrections for multiple comparisons, in: Neil. Salkind (Ed.), Encyclopedia of Measurement and Statistics, Sage, Thousand Oaks, CA, 2007, pp. 1–9.
[58] D. Olson, D. Delen, Advanced Data Mining Techniques, first ed., Springer, 2008.
[59] J.A. Hanley, B.J. McNeil, The meaning and use of the area under a receiver operating characteristic (ROC) curve, Radiology 143 (1) (1982) 29–36.
[60] I. Samoladas, S. Bibi, I. Stamelos, G.L. Bleris. Exploring the quality of free/open source software: a case study on an ERP/CRM system, in: 9th Panhellenic Conference in Informatics, Thessaloniki, Greece, 2003.
[61] I. Samoladas, G. Gousios, D. Spinellis, I. Stamelos, The SQO-OSS quality model: measurement based open source software evaluation, Open Source Development, Communities and Quality 275 (2008) 237–248.
[62] D. Spinellis, G. Gousios, V. Karakoidas, P. Louridas, P.J. Adams, I. Samoladas, I. Stamelos, Evaluating the quality of open source software, Electronic Notes in Theoretical Computer Science 233 (2009) 5–28.