

Received November 19, 2020, accepted December 9, 2020, date of publication December 21, 2020, date of current version December 31, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3046109

Measuring Class Cohesion Based on Client Similarities Between Method Pairs: An Improved Approach That Supports Refactoring

MUSAAD ALZHRANI 

Department of Computer Science, Albaha University, Albaha 65799, Saudi Arabia

e-mail: malzahr@bu.edu.sa

ABSTRACT Class cohesion is an important quality attribute that has an impact on other quality attributes such as understandability, testability, and maintainability. Therefore, classes with low cohesion should be refactored in order to improve their overall qualities. Many cohesion metrics have been introduced in the literature to automatically assess the quality of the class and support refactoring activities. Most existing metrics measure the class cohesion based on how the methods of a class are internally related to each other, while a few metrics measure the class cohesion based on how the methods are externally used by the clients of the class. Unfortunately, the existing client-based cohesion metrics cannot automatically support refactoring techniques such as the Extract Class refactoring. Therefore, this article proposes a new client-based cohesion metric that can be used to automatically identify Extract Class refactoring opportunities. The proposed metric is theoretically evaluated by proving the compliance of the metric to the mathematical cohesion properties, while it is empirically evaluated by conducting a large case study on three systems to compare the metric with other cohesion metrics. Finally, the paper introduces and evaluates an Extract Class refactoring approach based on the proposed cohesion metric.


INDEX TERMS Software engineering, software measurement, cohesion metrics, code refactoring.

I. INTRODUCTION

Software developers strive hard to design classes with high quality. Class cohesion is one of the most important quality attributes that influences the maintainability of the class [1], [2]. Classes with high cohesion are easier to understand and test, and thus easier to maintain. In addition, they have one responsibility and one reason to change [3]. Class cohesion is defined as the degree to which the elements of the class are related to each other [4], [5]. To avoid subjectivities and opinions about class cohesion and for the purpose of automation, many researchers have proposed objective metrics that tried to measure the class cohesion algorithmically. However, there is no universal agreement about a metric as a standard cohesion metric because the concept of cohesion is broad and leaves room for different interpretations of cohesion. Existing approaches for measuring class cohesion can be classified based on the view of the class during the measurement process into two categories: 1) approaches that

consider only the internal view of the class and 2) approaches that consider the internal and external view of the class. Most of existing class cohesion metrics (e.g., [6]–[10]) fall into the first category in which class cohesion is measured based on information extracted from the internal elements (methods and attributes) of the class regardless of the other classes in the system. For example, the Lack of Cohesion in Methods (*LCOM2*) [6] measures the class cohesion based on the pairs of methods that reference common attributes in their bodies. Cohesion metrics that fall in the first category have been used in many studies to support several refactoring techniques, most commonly the Extract Class refactoring (e.g., [11]–[13]). On the other hand, approaches in the second category measure the class cohesion based on information extracted from the elements of the class and its client classes where class *A* is considered as a client of class *B* if class *A* uses an element (a method or attribute) in class *B*. Only a few metrics fall into this category and these metrics have not been exploited in refactoring activities.

The client-based class cohesion metric (*CCC*) introduced in [1] and [14] is an example of metrics that measure the

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaobing Sun .

cohesion of class based on the internal and external view of the class. The metric measures the cohesion of the class based on the usage of its public methods by its clients. The idea behind *CCC* is that the cohesion of the class is judged by its clients. If a client uses all the public methods of the class, then they all contribute to a single responsibility from the perspective of that client and thus the class is considered to be cohesive. Otherwise, if the client uses only a subset of the public methods in the class, then the subset of the used public methods and the subset of the unused public methods by the client contribute to different responsibilities and therefore the class is not cohesive from the point of view of that client. *CCC* was theoretically evaluated using cohesion properties [15] and empirically evaluated by showing its relationship with testability [14] and maintainability [1]. However, the metric has the following limitations: 1) It does not automatically support refactoring activities such as the Extract Class refactoring because it cannot automatically determine the methods in the class that have low or no relationships because the metric simply measures the cohesion of the class based on the number of used methods in the class per each client of the class regardless the similarities or relationships between the methods of the class. 2) The metric considers only the public methods in the cohesion measurement and ignores the non-public methods in the class such as private and protected methods.

To overcome the above limitations, we propose in this article a new variation of *CCC*. The new metric measures the cohesion of a class based on the similarities (in terms of clients' usage) between each pair of methods including the non-public methods. We refer to this kind of similarity as client similarity between methods and we call the proposed metric *CCSM* which stands for Cohesion based on Client Similarity between Methods. By computing the client similarity between each pair of methods in the class, we can algorithmically identify the methods that have low/high client similarity. Thus, *CCSM* can automatically support refactoring techniques such as the Extract Class refactoring by automatically identifying the set of methods that can be extracted from a given class and put together into a separate class based on their client similarities. The paper presents a theoretical and empirical evaluation for *CCSM*. Theoretically, we show the compliance of *CCSM* to the cohesion properties defined in [15]. Empirically, we examine the correlations between 5 cohesion metrics including *CCSM* based the cohesion data of 1522 classes extracted from three open source systems for the purpose of comparing *CCSM* to the other cohesion metrics. In addition, we present an approach and a case study to show how *CCSM* can be used to automatically support the Extract Class refactoring. Our results indicate that *CCSM* is potentially useful and may offer benefits not offered by the other metrics. The novelty of this work lies on exploiting the client based cohesion in the (semi) automation of the Extract Class refactoring activities, which has not been studied before.

The contribution of the paper can be summarized as follows:

- introducing a new client-based cohesion metric (i.e., *CCSM*) that can automatically support refactoring activities,
- presenting an extensive theoretical and empirical evaluation of *CCSM*,
- introducing and evaluating an approach based on *CCSM* that can automatically identify Extract Class refactoring opportunities in a given class.

The rest of the paper is organized as follows. Section II discusses related work. Section III introduces the proposed metric. The theoretical and empirical evaluations of the proposed metric are given in Section IV and Section V, respectively. Section VI presents an Extract Class refactoring approach based on the proposed metric. Finally, the conclusion and future work is given in Section VII.

II. RELATED WORK

Many approaches have been introduced in the literature for the measurement of class cohesion. These approaches can be classified based on the view of the class during the measurement time into two sets: 1) internal view-based approaches, and 2) internal and external view-based approaches. In the following we discuss and give examples of each set.

A. INTERNAL VIEW-BASED APPROACHES

In these approaches only the internal elements of the class are considered during the process of cohesion measurement. The majority of existing approaches for measuring the class cohesion are internal view-based approaches. Class cohesion in these approaches is mostly measured based on the similarities (in terms of referenced attributes) between each pair of methods in the class. The intuition behind these approaches is that methods that access attributes in common are related to each other because they depend on the same data. Thus, the more methods share attributes in the class, the more cohesive the class is. An example of these approaches is the approach by Chidamber and Kemerer in [6]. They introduced the Lack of Cohesion in Methods (*LCOM2*) metric which measures the cohesion of the class according to the following definition:

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

where P is the number of non-similar pairs of methods and Q is the number of similar pairs where a pair of methods is non-similar if the two methods do not reference common attributes and a pair of methods is similar if both methods reference at least one common attribute. The value of the metric is not normalized; it ranges from 0 to the total number of the pairs of methods in the class. *LCOM2* is an inverse cohesion metric because it measures the lack of cohesion which means the higher the value of *LCOM2* for a class, the lower the cohesion of the class and vice versa.

In a similar manner, Bieman and Kang [7] defined the class cohesion metric Tight Class Cohesion (*TCC*) which measures the cohesion of a class based on the number of pairs of public methods in the class that share common attributes. *TCC* is defined as follows:

$$TCC = \frac{PPM}{NP}$$

where *PPM* is the number of pairs of public methods that directly or transitively share an attribute in the class and *NP* is the total number of pairs of public methods in the class. A pair of public methods directly shares an attribute if the two methods reference the attribute and transitively shares an attribute if one of the two methods does not reference the attribute but it directly or transitively calls a method that references the attribute.

The above two approaches [6], [7] do not consider the number of shared attributes between a pair of methods when calculating the similarity between the pair meaning that a pair of methods that shares only one attribute has the same similarity degree as another pair of methods that shares all the attributes of the class. To address this limitation, Bonja and Kidanmariam [8] introduced the Class Cohesion (*CC*) metric which measures the cohesion of a class based on the degree of similarity between the pairs of methods in the class. The degree of similarity between method *i* and *j* is defined as follows:

$$Similarity(i, j) = \frac{|A_i \cap A_j|}{|A_i \cup A_j|}$$

where A_i is the set of attributes referenced by the method *i* and A_j is the set of attributes referenced by the method *j*. Then the cohesion of the class is calculated as the average of similarities between all the pairs of methods in the class.

Al Dallal and Briand [10] proved that *CC* do not satisfy all of the key cohesion properties defined in [15]. *CC* namely violates the Monotonicity property which holds that the addition of similarities between the pairs of methods can not decrease the class cohesion. To address this limitation, they redefined the degree of similarity between the method *i* and *j* as follows:

$$Similarity(i, j) = \frac{|A_i \cap A_j|}{|L|}$$

where A_i and A_j are the set of attributes referenced by the method *i* and *j*, respectively and L is the set of attributes in class. Similar to *CC*, they defined the Low-level design Similarity-based Class Cohesion (*LSCC*) as the average of similarities between all the pairs of methods in the class.

Different from all the previous approaches, Marcus *et al.* [9] introduced the Conceptual Cohesion of Classes (*C3*) metric which calculates the cohesion of a class as the average of the conceptual similarities between each pair of methods in the class. The advanced information retrieval technique Latent Semantic Indexing is used to measure the degree of conceptual similarity between the methods of the class. The value of *C3* ranges from 0 to 1 where 0 means the class has

no conceptual cohesion and 1 means the class has fully conceptual cohesion.

B. INTERNAL AND EXTERNAL VIEW-BASED APPROACHES

A few approaches of class cohesion measurement do not only consider the internal elements of the class but also consider how the elements of the class are used externally by its clients. These approaches known as client-based approaches. The intuition behind them is that if a client uses the all the considered elements (methods or attributes) of class, then the elements of the class are functionally related to each other and they contribute to one responsibility from the perspective of that client. Thus, the more elements of the class are used by the clients of the class, the more cohesive the class is. Lack of Coherence in Clients (*LCIC*) introduced in [16] is one of these metrics. The cohesion of a class is measured by *LCIC* as the average of the lack of coherence in all the clients of the class where the lack of coherence per a client is calculated as one minus the ratio of the number attributes used by the client to the number of attributes in the class that the client can access. Formally, *LCIC* is defined as follows:

$$LCIC = \begin{cases} \sum_{x \in C} \frac{(1 - \frac{|U|}{|A|})}{|C|} & \text{if } |C| > 0, \\ 1 & \text{otherwise} \end{cases}$$

where C is the set of the clients of the class, A is the set of the attributes in the class that client x can access, and U is the set of the used attributes in the class by client x . A key limitation of *LCIC* is that it does not consider the relationship between the methods of class nor how they are used by the clients. If there is a client that uses one method in the class and that method references all the attributes in the class, then the client uses all the attributes of the class and thus the class is considered to be fully cohesive from the perspective of that client regardless the other methods of the class. Moreover, the responsibility of the class is usually determined through its the methods. Thus, it is important consider the methods of the class and their relationships when measuring the class cohesion.

Alzahrani and Melton [14] proposed the Client-based Class Cohesion (*CCC*) metric that overcomes the limitation of *LCIC*. *CCC* measures the cohesion of a class based on the clients' usage of the public methods of the class. The definition of *CCC* is given by:

$$CCC = \begin{cases} \sum_{x \in C} \frac{(\frac{|UPM|}{|NPM|})}{|C|} & \text{if } |C| > 0, \\ 1 & \text{otherwise} \end{cases}$$

where C is the set of the clients of the class, UPM is the set of the public methods used by the client x and NPM is the set of the public methods in the class. However, as explained in the Introduction Section, *CCC* has two limitations: it does not automatically support refactoring techniques such as the Extract Class refactoring and it does not consider the non-public methods in the class.

In this article, we introduce the client-based cohesion metric *CCSM* that overcomes the two previously mentioned limitations of *CCC*. The main difference between *CCSM* and *CCC* is that *CCSM* measures the cohesion of a class based on the similarities between each pair of methods in a class whereas *CCC* measures the cohesion of a class based on the number of public methods used by each client of the class regardless the similarities between the methods. In addition, *CCSM* considers the non-public methods when measuring the cohesion of a class whereas *CCC* does not. The difference between our approach and the internal-view based approaches is in the way we compute similarities between method pairs. *CCSM* computes the similarity between two methods based on the number shared clients between the methods whereas the majority of the internal-view based approaches calculate the similarity between two methods based on the number of shared attributes.

III. COHESION BASED ON CLIENT SIMILARITY BETWEEN METHODS (CCSM)

The Cohesion based on Client Similarity between Methods (*CCSM*) is defined based on client similarity between each pair of methods in the class. In the following, we first define our model for an object-oriented system. We next define the client similarity between methods and present the definition of the proposed cohesion metric (*CCSM*).

A. MODEL DEFINITION

Definition 1 (System and Classes): An object oriented system S consists of a set class $S = \{c_1, c_2, \dots, c_n\}$ where n is the number of classes in the system S .

Definition 2 (Methods of a Class): Each class $c \in S$ has a set of methods $M(c) = \{m_1, m_2, \dots, m_k\}$ where k is the number of the methods in the class c .

Definition 3 (Public and Non-Public Methods of a Class): Each class $c \in S$ has a set of public methods $M_{pub}(c)$ and a set of non-public methods $M_{npub}(c)$ such that $M_{pub}(c) \cap M_{npub}(c) = \phi$ and $M_{pub}(c) \cup M_{npub}(c) = M(c)$.

The public methods of the class $M_{pub}(c)$ are the methods that can be called from any class in the system.

The non-public methods of the class $M_{npub}(c)$ are the methods that can be only called from certain classes in the system. For example, private methods in the class c can be called from the class c and no other class in the system can call them.

Definition 4 (Clients of a Method): Each method $m \in M(c)$ has a set of clients $Clients_M(m) = \{client_1, client_2, \dots, client_x\}$ where x is the number of clients the method m has and $client_j$ is any class in system excluding the class c (i.e., $client_j \in S - c$) that is a direct or indirect client of m (see Definition 5).

Definition 5 (Direct and Indirect Clients of a Method): Each method $m \in M(c)$ has a set of direct clients $Clients_{MDir}(m)$ and a set of indirect clients $Clients_{MIndir}(m)$ such that $Clients_{MDir}(m) \cup Clients_{MIndir}(m) = Clients_M(m)$.

The direct clients of the method m are the classes (excluding the class c) that have a direct static or polymorphic call

for the method m . Some of non-public methods (e.g., private methods) in the class c cannot have direct clients because they cannot be called from any other class in the system.

The indirect clients of the method m are the classes that are direct clients of another method m' in the class c such that the method m' directly or indirectly calls the method m . Any non-public method in the class c can have indirect clients as they can be called from any other method in the class c .

Definition 6 (Clients of a Class): The class c has a set of clients $Clients_C(c) = \{client_1, client_2, \dots, client_y\}$ which is the union of $Clients_M(m)$ for each $m \in M(c)$. That is $Clients_C(c) = \bigcup_{m \in M(c)} \{Clients_M(m)\}$.

B. METRIC DEFINITION

The proposed metric measures the cohesion of a class based on the degree of client similarity between each pair of methods in the class.

Definition 7 (The Degree of Client-Based Similarity Between Two Methods): Let $c \in S$ and $m_i, m_j \in M(c)$. Then the degree of client-based similarity between the method m_i and the method m_j is defined by:

$$Sim_{clients}(m_i, m_j) = \begin{cases} \frac{|Clients_M(m_i) \cap Clients_M(m_j)|}{|Clients_C(c)|} & \text{if } |Clients_C(c)| > 0, \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Definition 8 (Cohesion Based on Client Similarity Between Methods of a Class (CCSM)): The cohesion based on client similarity between the methods of class c is defined as average of the degrees of client similarities between each pair of methods in the class c and it is formally given by:

$$CCSM(c) = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^k Sim_{clients}(m_i, m_j) \quad (2)$$

where k is the number of methods in the class c .

C. AN EXAMPLE OF CCSM

We give a simple example to better understand how *CCSM* can be applied to measure the cohesion of a class. Consider class A shown in Fig. 1. In order to measure the *CCSM* for the class A , we need first to extract its set of methods which is the following:

$$M(A) = \{a1, a2, a3, a4\}.$$

We next extract the clients of each method in the class.

$$\begin{aligned} Clients_M(a1) &= \{B\}.^1 \\ Clients_M(a2) &= \{B\}. \\ Clients_M(a3) &= \{B\}. \\ Clients_M(a4) &= \{C\}. \end{aligned}$$

After we have extract the clients of each method in the class A , we can extract the clients of the class which is the union of

```

public class A {
    ...
    private void a1()
    {
        ...
    }
    public void a2()
    {
        a1();
        ...
    }
    public void a3()
    {
        a1();
        ...
    }
    public void a4()
    {
        ...
    }
}

public class B {
    A objA;
    ...
    public void b1()
    {
        objA.a2();
        ...
    }
    public void b2()
    {
        objA.a3();
        ...
    }
}

public class C {
    A objA;
    ...
    public void c1()
    {
        objA.a4();
        ...
    }
}
    
```

FIGURE 1. Three hypothetical classes.

TABLE 1. The degree of client similarity between each pair of methods in the class A.

	a2	a3	a4
a1	0.5	0.5	0
a2		0.5	0
a3			0

all the clients of its methods.

$$Clients_C(A) = \{B, C\}.$$

We next calculate the degree of client similarities between each pair of methods in the class using Equation (1), see Table 1.

Finally, we calculate the *CCSM* of the class A using Equation (2).

$$CCSM(A) = \frac{2}{4(4-1)} \times (0.5 + 0.5 + 0 + 0.5 + 0 + 0)$$

$$CCSM(A) = 0.25$$

IV. THEORETICAL EVALUATION OF CCSM

Briand *et al.* [15] defined four mathematical properties for cohesion metrics. Cohesion metrics that satisfy these properties are expected to be better quality indicators. In the following, we prove *CCSM* satisfy all of these properties.

Property 1 (Non-Negativity and Normalization): This property holds that the value of a cohesion metric for a class is not negative and normalized. *CCSM* clearly satisfies the Non-Negativity and Normalization property because the minimum value of *CCSM* is 0 and the maximum value of

¹Class B is an indirect client of the private method a1 because Class B calls the method a2 which in turn calls the method a1

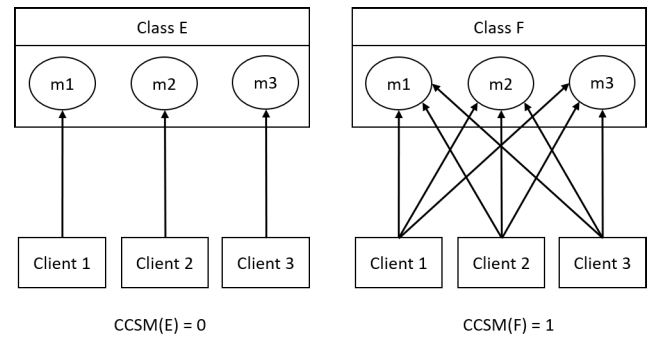


FIGURE 2. Two hypothetical classes E and F. Each pair of methods in the class E has 0 degree of client similarity as there is no two methods used by a common client whereas the degree client similarity between each pair of methods in class F is 1 as each method is used by the same set of clients.

it is 1. The value of *CCSM* is 0 when the methods of the class have disjoint sets of clients because the degree of the client similarity between each pair of methods will be 0 using Equation (1) (see class E in Fig. 2 as an example). The value of *CCSM* is 1 when all the methods of the class have the same set of clients because the degree of client similarity between every pair of methods in the class will be 1 using Equation (1) (see class F in Fig. 2). Thus, the interval of *CCSM* is [0, 1] and therefore the metric satisfies Property 1.

Property 2 (Null and Maximum Value): This property holds that the value of a cohesion metric for a class is null if there are no relationships between the elements of the class, and the value of a cohesion metric for a class is the maximum if all the possible relationships between the elements of the class are present. In case of *CCSM*, two methods have a relationship (i.e., some degree of client similarity) if they share common clients. The value of *CCSM* is 0 (null) when there is no relationship between any pair of methods in the class (i.e., the degree of client similarity between each pair of methods is 0, see class E in Fig. 2) and the value of *CCSM* is 1 (the maximum) when each pair of methods in the class have a full relationship by sharing all clients of the class (i.e., the degree of client similarity between each pair of methods is 1, see class F in Fig. 2 for example). Therefore, *CCSM* satisfies the Null and Maximum Value property.

Property 3 (Monotonicity): This property holds that the addition of relationships between the elements of a class cannot decrease the cohesion of the class. Given a class *c*, let $|Clients_C(c)| > 0$ and $m_i, m_j \in M(c)$. Let $|Clients_M(m_i) \cap Clients_M(m_j)| = 0$. Then the methods m_i and m_j have no relationship because they do not share a client, which means the degree of client similarity between the two methods is 0. The addition of a relationship between the methods m_i and m_j means adding one of existing clients of the class to the set of clients of each method (i.e., $\exists client_s \in Clients_C(c)$ and we add $client_s$ to $Clients_M(i)$ and $Clients_M(j)$ such that $|Clients_M(m_i) \cap Clients_M(m_j)| > 0$). When this addition occurs, the value of *CCSM* will never decrease. In fact, it will increase because this addition of relationship between the methods m_i and m_j will increase degree of client similarity

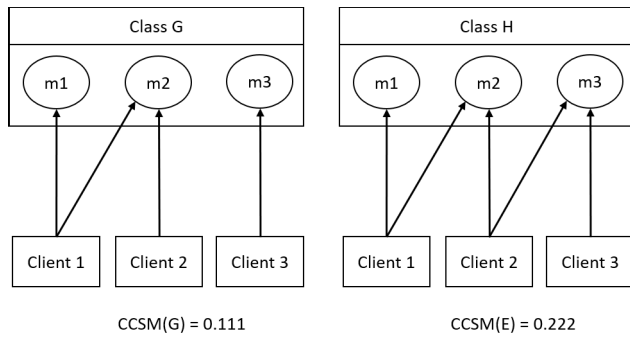


FIGURE 3. Methods m_2 and m_3 has no relationship in class G as they are not used by a common client. A new relationship between the two methods is added in class H by making client2 use the method m_3 .

between the two methods and it will never decrease it between any other two methods in the class. Therefore, $CCSM$ satisfies Monotonicity property.

For example, consider the classes G and H shown in Fig. 3. The methods m_2 and m_3 in class G has no relationship because they are not used by a common client. When we added a new relationship between the two methods in the class H the value of $CCSM$ for the class H increased from 0.111 to 0.222.

Property 4 (Cohesive Modules): This property holds that when two unrelated classes A and B are merged into one Class M , the cohesion of the resulting class M cannot be more than the maximum cohesion of the original A and B .

Let $m_i, m_j \in M(A)$ and k be the number of methods in class A . Then:

$$CCSM(A) = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^k Sim_{clients}(m_i, m_j)$$

By substituting $Sim_{clients}(m_i, m_j)$ with Equation (1), we get:

$$CCSM(A) = \frac{2 \sum_{i=1}^{k-1} \sum_{j=i+1}^k |Clients_M(m_i) \cap Clients_M(m_j)|}{k(k-1)|Clients_C(A)|}$$

For simplicity, let $a = 2 \sum_{i=1}^{k-1} \sum_{j=i+1}^k |Clients_M(m_i) \cap Clients_M(m_j)|$ and $b = |Clients_C(A)|$. Then:

$$CCSM(A) = \frac{a}{k(k-1)b}$$

Similarly with class B , let $m'_i, m'_j \in M(B)$ and l be the number of methods in class B . Then

$$CCSM(B) = \frac{2 \sum_{i=1}^{l-1} \sum_{j=i+1}^l |Clients_M(m'_i) \cap Clients_M(m'_j)|}{l(l-1)|Clients_C(B)|}$$

Also, for simplicity, let $c = 2 \sum_{i=1}^{l-1} \sum_{j=i+1}^l |Clients_M(m'_i) \cap Clients_M(m'_j)|$ and $d = |Clients_C(B)|$. Then:

$$CCSM(B) = \frac{c}{l(l-1)d}$$

In our case of $CCSM$, the two classes A and B are unrelated if there does not exist a method in class A that shares a common client with a method in class B (i.e., $\nexists m \in M(A)$ and

$\nexists m' \in M(B)$ such that $|Clients_M(m) \cap Clients_M(m')| > 0$). This implies that:

$$\begin{aligned} CCSM(M) &= \frac{a+c}{(k+l)(k+l-1)(b+d)} \\ &= \frac{a+c}{(k^2+l^2+2kl-k-l)(b+d)} \end{aligned}$$

Assume that $CCSM(A) \geq CCSM(B)$, then:

$$\begin{aligned} \frac{a}{k(k-1)b} &\geq \frac{c}{l(l-1)d} \implies \frac{al(l-1)d}{k(k-1)b} \geq c \\ &\implies \frac{al(l-1)d}{k(k-1)b} + a \geq c + a \\ &\implies \frac{al(l-1)d + ak(k-1)b}{k(k-1)b} \geq c + a \\ &\implies \frac{a(l(l-1)d + k(k-1)b)}{k(k-1)b} \geq c + a \\ &\implies \frac{a}{k(k-1)b} \geq \frac{c+a}{l(l-1)d + k(k-1)b} \\ &\implies \frac{a}{k(k-1)b} \geq \frac{c+a}{(l^2-l)d + (k^2-k)b} \\ &> \frac{a+c}{(k^2+l^2+2kl-k-l)(b+d)} \\ &\implies CCSM(A) > CCSM(M) \end{aligned}$$

Therefore, $CCSM$ satisfies the Cohesive Modules property.

To better understand how $CCSM$ satisfies the Cohesive Modules property, consider the classes I, J and M shown in Fig. 4. The classes I and J are unrelated because there is no method in class I that shares a client with a method in class J. The value of $CCSM$ for the class J is 0.833 which higher than the value of $CCSM$ for the class I ($CCSM(I) = 0.333$). When we merge the two classes I and J into one class, the resulting class will be the class M. The $CCSM$ of the resulting class M is 0.233 which is less than value of $CCSM$ of the class J.

V. EMPIRICAL EVALUATION

We conduct a case study to empirically evaluate $CCSM$ and compare it with other well-known class cohesion metrics based on classes extracted from real systems.

A. CONSIDERED SYSTEMS AND CLASSES

We selected three systems in our case study namely: JHotDraw version 9.0 [17], ArgoUML version 0.34 [18], and Xerces2 version 2.12.0 [19]. JHotDraw is a two-dimensional graphics framework for structured drawing editors. ArgoUML is a UML modeling tool that supports all standard UML diagrams. Xerces2 is a library for parsing, validating and manipulating XML documents. We selected these systems based on the following criteria: the systems are open source, implemented in Java, and from different domains. In addition to meeting the above criteria, we considered these three systems because they have been used in several well-known studies in the field (e.g., [10], [12], [13]).

Table 2 reports the number of classes we consider from each system. We select any class that meets the following

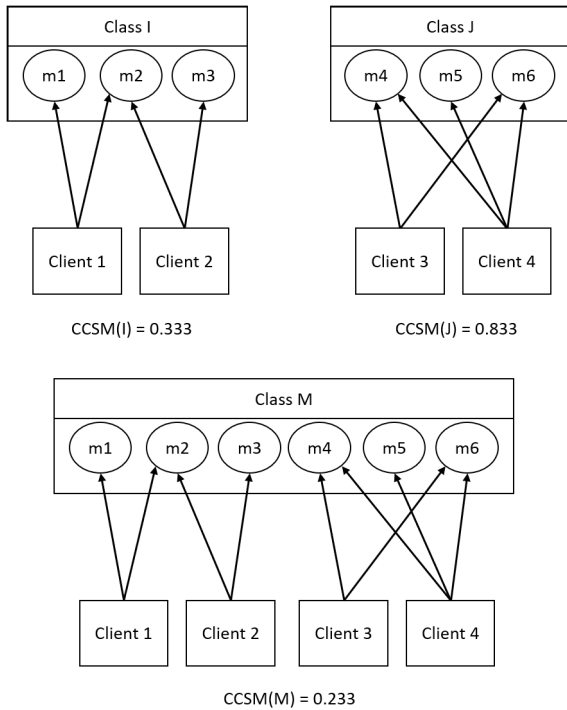


FIGURE 4. Class M is resulted from merging the two unrelated classes I and J. Classes I and J are unrelated because the methods in class I share no clients with methods in class J.

TABLE 2. Number of considered classes from each system.

System	Number of Studied Classes
JHotDraw 9.0	341
ArgoUML 0.34	783
Xerces2 2.12.0	398

requirements: the class has at least two methods, one attribute and one client. The reason of setting these requirements is to have a defined value for each considered metric and to avoid special cases for the considered metric (e.g., the case of having a class with no clients for the metric *CCSM*). The total number of the considered classes from the three systems is 1522.

B. CONSIDERED METRICS

We select the following metrics in our case study: *LCOM2*, *LSCC*, *CCC*, *#Methods* and *CCSM*. *LCOM2* and *LSCC* are internal view-based class cohesion metrics. We considered these two metrics because they are well-known metrics that have been studied and evaluated previously in the literature (e.g., [2], [10], [20], [21]). *CCC* is a client-based class cohesion metric and we considered it because the newly proposed metric *CCSM* is a variation of it and we want compare both metrics to each other. *#Methods* is a size metric that counts the number of methods in the class. We considered the size metric *#Methods* because size is an important characteristic of the class that has an impact on other quality attributes of the class such as fault-proneness and we wanted to investigate the relationship between the considered cohesion metrics and the size.

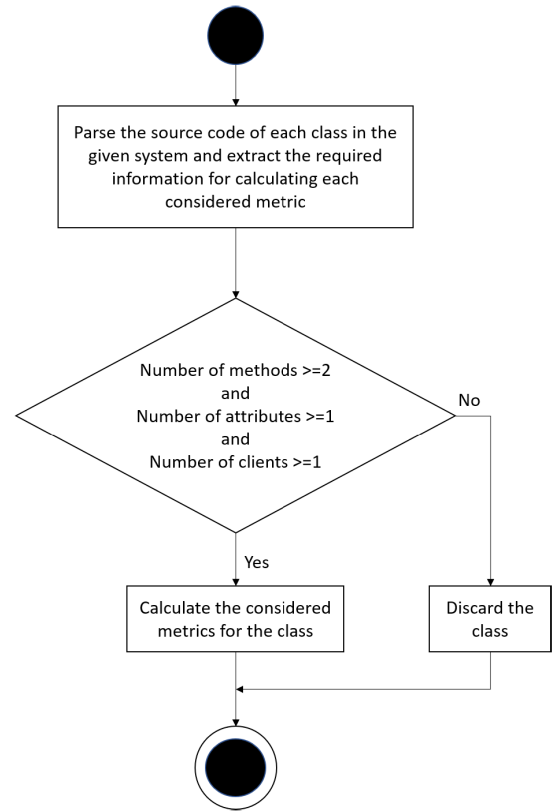


FIGURE 5. The main steps for calculating the considered metrics by our Java tool.

C. PLANNING

The main purpose of our case study is investigate the relationship between the proposed cohesion metric *CCSM* and the other considered metrics. For this purpose, we first compute set of descriptive statistics for each considered metric including the minimum value, first quartile, median, third quartile, maximum value, mean, and standard deviation to give an overall picture of the differences and similarities among the considered metrics. In addition, we calculate Pearson’s correlation coefficients between each pair of the considered metrics. The value of a correlation coefficient which ranges from -1 to $+1$ reflects the strength of the linear relationship between the two metrics. A value of -1 indicates a perfect negative correlation between the two metrics and a value of $+1$ indicates a perfect positive correlation between the two metrics. If the value of correlation coefficient is 0, then there is no linear relationship between the two metrics. Similar to [22], in this case study, we consider correlation of 0.1 to be trivial, 0.1 – 0.3 minor, 0.3 – 0.5 moderate, 0.5 – 0.7 large, 0.7 – 0.9 very large, and 0.9 – 1 almost perfect.

D. TOOLS

We developed our own Java tool based JavaParser [23] to automatically compute the considered metrics. The main steps for computing the considered metrics by our tool is depicted in Fig. 5. Our tool parses the source code of each class in the given system and extract the required information

TABLE 3. Descriptive statistics for the considered metrics based on the 341 classes selected from JHotDraw 9.0.

Metric	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean	Std. Dev.
#Methods	2	5	8	12.67	80	16	12.43
LCOM2	0	1	14	109.6	2734	61	309.01
LSCC	0	0.003	0.033	0.103	1	0.1	0.189
CCC	0	0.041	0.117	0.259	1	0.333	0.325
CCSM	0	0.017	0.049	0.149	1	0.167	0.234

TABLE 4. Descriptive statistics for the considered metrics based on the 783 classes selected from ArgoUML 0.34.

Metric	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean	Std. Dev.
#Methods	2	3	6	14	409	12.52	26.295
LCOM2	0	1	9	55	83344	373.2	4210.478
LSCC	0	0	0.015	0.076	1	0.099	0.215
CCC	0	0	0.105	0.5	1	0.297	0.372
CCSM	0	0.003	0.067	0.322	1	0.228	0.317

TABLE 5. Descriptive statistics for the considered metrics based on the 398 classes selected from Xerces2 2.12.0.

Metric	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean	Std. Dev.
#Methods	2	4	9	19	125	15.69	18.383
LCOM2	0	0	14	90.5	7220	195.9	704.611
LSCC	0	0.005	0.04	0.199	1	0.17	0.275
CCC	0	0	0.17	0.342	1	0.282	0.329
CCSM	0	0.02	0.067	0.189	1	0.166	0.245

for calculating the considered metrics. For example, we need to know the clients of each method in the class in order to calculate the proposed metric *CCSM*. The tool automatically counts the number of methods, attributes and clients of the class. If the number of methods is greater than or equal to 2 and number attributes is greater than or equal to 1 and the number of clients is greater than or equal to 1, the tool computes the considered metrics for the class and reports the results in a file. Otherwise, the class is discarded. At the end, the tool generates one CSV file that includes the values of the metrics for all the considered classes in the given system.

We used the free software project R [24] to compute the descriptive statistics for the considered metrics and the Pearson's correlation coefficients between them based on the results generated by our Java tool.

E. RESULTS AND DISCUSSION

Tables 3, 4, and 5 report descriptive statistics for the considered systems. The results show that the minimum value of the cohesion metrics *CCSM*, *CCC*, and *LSCC* is 0 and the maximum value of the these metrics is 1. As we explained in our theoretical evaluation of *CCSM* in Section IV, the value of *CCSM* is 0 when the methods of the class in question have disjoint sets of methods and the value of the metric is 1 when all the methods have the same set of clients. The value of *CCC* is 0 when each client in the class uses only one public method in the class (refer to [14] for more details) and the value of the metric is 1 when each client in the class uses all the public methods of the class. For *LSCC*, the value of the metric is 0 when the class does not have a pair of methods that reference a common attribute in the class and the value of the metric is 1 when each method in the class references all the attributes of the class.

The mean and quartiles of *CCSM*, *CCC*, and *LSCC* are relatively small which indicates that the values of these metrics are relatively small for most the considered classes across the three systems. The reason behind the small values of *CCSM* and *CCC* is that they are affected by the methods of the class that are not used by the clients of the class. In the case of *CCC*, in order for a class to be fully cohesive, all the clients of the class must use all the public methods of the class which means all the public methods of the class share all the clients of the class. Similarly for *CCSM*, in order for a class to be fully cohesive, all the methods in the class must share all the clients of the class. Ideally, it is better if all the methods of the class share all clients of the class because this means the methods of have a strong relationship and they all contribute to a single responsibility from the perspective of the clients of the class. In reality, classes usually have core methods that are used by most of the clients and utility methods that are only used by some of the clients which leads to smaller values of *CCSM* and *CCC*. For *LSCC*, all the methods of the class must reference all the attributes in the class in order for a class have a full cohesion value. In practice, classes usually have methods that only reference some of the attributes in the class which leads to smaller values of *LSCC*. An example of these methods are the getters² and the setters³ in the class. A getter or setter usually references only one attribute in the class.

We can observe that *LCOM2* has extremely greater mean, median, and maximum values compared to the other cohesion metrics. This is because the *LCOM2* is not normalized (i.e., it does not have an upper bound), whereas the values of the

²A getter is a method that is responsible for reading the value of an attribute in the class

³A setter is a method that is responsible for updating the value of an attribute in the class.

TABLE 6. Pearson's correlations between the metrics for the classes selected from JHotDraw 9.0.

	#Methods	LCOM2	LSCC	CCC	CCSM
#Methods	1	0.868	-0.195	-0.277	-0.139
LCOM2		1	-0.165	-0.169	-0.048
LSCC			1	0.022	-0.031
CCC				1	0.605
CCSM					1

TABLE 7. Pearson's correlations between the metrics for the classes selected from ArgoUML 0.34.

	#Methods	LCOM2	LSCC	CCC	CCSM
#Methods	1	0.857	-0.053	-0.133	-0.14
LCOM2		1	-0.038	-0.055	-0.053
LSCC			1	-0.007	-0.041
CCC				1	0.64
CCSM					1

TABLE 8. Pearson's correlations between the metrics for the classes selected from Xerces2 2.12.0.

	#Methods	LCOM2	LSCC	CCC	CCSM
#Methods	1	0.838	-0.291	-0.15	-0.159
LCOM2		1	-0.165	-0.097	-0.077
LSCC			1	-0.076	-0.1
CCC				1	0.578
CCSM					1

other cohesion metrics range from 0 to 1, inclusively. The minimum value of *LCOM2* is 0 and this occurs when the number pairs of methods that share an attribute is greater than number of pairs of method that do not share an attribute.

Finally, the results in Tables 3, 4, and 5 indicate that *CCSM* has in general smaller values than *CCC*. The reason behind this is that *CCSM* computes the cohesion of the class based on the client similarities between each pair of methods in the class, whereas *CCC* computes the cohesion of a class based on the number of used public methods in the class by the clients of class regardless the client similarities between the methods of class. As a result, the value of *CCSM* will be greatly and negatively affected by the number of pairs of methods that do not share common clients compared to *CCC*. This leads to having smaller values of *CCSM* compared to the values of *CCC* in most cases. The standard deviation of both metrics is relatively similar.

Tables 6, 7, and 8 present the Pearson's correlation coefficients between each pair of the considered metrics across the three systems. The results show that the correlation between the cohesion metric *LCOM2* and the size metric *#Methods* is very large. The reason behind the very large correlation between these two metrics is that *LCOM2* is greatly influenced by the number of methods in the class because the metric is not normalized.

CCSM has trivial correlations with the internal view-based cohesion metrics *LCOM2* and *LSCC* and the size metric *#Methods* across the three systems. The trivial correlations between *CCSM* and the internal view-based cohesion metrics *LCOM2* and *LSCC* indicate that *CCSM* addresses a different aspect of cohesion that is not addressed by the internal view-based cohesion metrics. This was expected because

CCSM measures the cohesion of a class based on the shared clients of the class between the methods of the class, whereas the internal view-based metrics measure the cohesion based on the shared attributes between the methods. The correlation between *CCSM* and *CCC* is moderate across the three systems because both metric measure the cohesion based on the clients of the class.

The correlations between *LSCC* and *LCOM2* is trivial across the three systems even though both metrics measure the cohesion based on the shared attributes between the methods. Two reasons behind this. First, *LCOM2* is not normalized. Second, *LSCC* considers the number of shared attributes when measuring the degree of similarity or relationship between two methods, whereas *LCOM2* does not. If we have a class that has 4 attributes and there are two methods in the class that share one attribute, then the degree of similarity between the two methods is $\frac{1}{4}$ in the case of *LSCC*, whereas the degree of similarity between the two methods is 1 in the case of *LCOM2* because *LCOM2* considers two methods to be fully similar if they share at least one attribute in the class.

The directions of correlations between the three cohesion metrics *CCSM*, *CCC*, and *LSCC* and the cohesion metric *LCOM2* is negative because the *LCOM2* is an inverse cohesion metric meaning that higher values of *LCOM2* indicate lower cohesion, whereas *CCSM*, *CCC*, and *LSCC* measure the cohesion directly. Also the directions of correlations between *CCSM*, *CCC*, and *LSCC* and the size metric *#Methods* is negative because usually the class becomes less cohesive when its size increases. *LCOM2* has positive correlation with *#Methods* because as we explained previously the *LCOM2* measures the lack of cohesion and larger classes tend to have higher lack of cohesion. The direction of the correlation between the *CCSM* and *LSCC* is negative. We expected these two metrics to have positive correlation because they both measure cohesion directly. However, the magnitude of the correlation between the these two metrics is extremely low. We can say there is no relationship between the two metrics. The reason behind this extremely low correlation between the two metric is that they measure different aspects of cohesion.

Overall, we find the results of the metric *CCSM* encouraging. The metric has trivial correlations with the internal view-based cohesion metrics and the size metric *#Methods*, which means the metric addresses properties of quality that are not addressed by other metrics. This is good because the metric can be used to complement the other internal-based cohesion metrics and the size metric *#Methods* when predicting external quality attributes that are believed to be influenced by cohesion and size such as fault-proneness, maintainability and testability. In addition, the *CCSM* has moderate correlation with *CCC*. This is also good because *CCC* was found to be usefull predictor for testability [14] and maintainability [1] and we expect *CCSM* to be also useful predictor for testability and maintainability since it has moderate correlation with *CCC*. However, we believe *CCSM* is more beneficial than *CCC* because it considers

all the methods of the class when measuring the cohesion, whereas *CCC* considers only the public methods. In addition *CCSM* can automatically support refactoring techniques such as Extract Class refactoring which can not be automatically supported by *CCC* as we will see in the next Section.

F. THREATS TO VALIDITY

Several factors may affect the results of our empirical evaluation and limit our interpretation. First, the selected three systems are open source systems and implemented in the same programming language (i.e., Java). In addition, the domains of selected systems may not be representative of all the domains in the software industry. Furthermore, the size and the number of classes considered from the the selected systems may not be representative for the classes of the systems in the software industry. As suggested by [10], we need to consider a set of systems from the software industry that are representative in terms of the size and the number of classes and in terms of domains; and implemented in different programming languages in order to generalize our results.

VI. USING CCSM TO SUPPORT EXTRACT CLASS REFACTORING

Extract Class refactoring refers to the process of splitting a class that has many responsibilities (known as a Blob or God Class) into a set of smaller classes, each of which has a single responsibility [25]. Cohesion is used to indicate whether a class has one responsibility or more because a highly cohesive class is believed to have only a single responsibility [13]. Therefore, many approaches (e.g., [11]–[13], [26]–[38]) have employed cohesion metrics to automatically support the Extract Class refactoring because performing manually the Extract Class refactoring costs much time and effort. In these approaches, the different responsibilities of a non-cohesive class are determined by identifying the methods in the class that have strong similarities or relationships. Each group of methods that have strong relationships is suggested to be extracted into a separate class. Only internal view-based cohesion metrics have been used to support the Extract Class refactoring. Therefore, we introduce an approach that uses the newly proposed client-based cohesion metric *CCSM* to support the Extract Class refactoring. The approach is an extension of our previous works [39], [40] on the Extract Class refactoring.

In the following, we first discuss the potential benefits of using client-based cohesion metrics in supporting the Extract Class refactoring. We next explain the limitation of the client-based cohesion metric *CCC* in supporting the Extract Class refactoring. Then we present the proposed approach that exploits *CCSM* to support the Extract Class refactoring and present an example of application for the proposed approach. Finally, we present an Extract Class refactoring case study.

A. THE BENEFITS OF USING CLIENT-BASED COHESION METRICS IN SUPPORTING THE EXTRACT CLASS REFACTORING

Client-based cohesion metrics can be potentially more beneficial in supporting the Extract Class refactoring than the cohesion metrics that consider only the internal view of the class for the following reasons: While internal view based metrics can only identify the methods that have low similarities or relationships with each other, client-based cohesion metrics can in addition to that identify the case of having a class that has low cohesion and many clients. More attention should be paid to that case because if the class has low cohesion and many clients, not only the maintenance effort of the class will increase but also the the maintenance effort of its clients will increase as they depend on the class. One the other hand, if the class has low cohesion and no clients or a few number clients, then the overall maintenance effort of the class and its clients will be low comparing to the previous case. Another benefit of using client-based cohesion metrics in the Extract Class refactoring is that it supports the adherence to the Interface Segregation Principle (ISP) which states that clients of a class should not be forced to depend on methods they do not use in the class [3]. ISP is an important design principle and failing to adhere to it can increase the overall maintenance of the system. For example, when a client of a class exerts changes to some methods in the class, the other clients of the class can be affected even if they do not use those methods [3]. Client-based cohesion metrics can be used to identify the methods of the class that are used by disjoint sets of clients. Those methods can be extracted into separate classes.

B. THE LIMITATION OF CCC IN SUPPORTING THE EXTRACT CLASS REFACTORING

The cohesion metric *CCC* cannot automatically identify the methods of a class that have high/low similarities (or relationships) because the metric measures the cohesion of the class based on the number of used methods by each client of the class and the metric does not compute the similarities between the methods. As a result, the metric cannot automatically support the Extract Class refactoring. For example, consider the class *A* and its clients represented in Fig. 6 where the circles represent the methods of the class *A* and the arrows represent the usage of the methods by the clients of *A*. It is clear that the methods *m1*, *m2* and *m3* have strong client similarity. Also, the methods *m4*, *m5* and *m6* have strong client similarity. If we split the class *A* into two classes where the first the class has the methods *m1*, *m2* and *m3* and the second class has the methods *m4*, *m5* and *m6*, we will have two classes with high cohesion and we will better adhere to ISP because we will not force the clients to depend on methods they do not use. The metric *CCC* cannot be used to automatically split the class *A* into two classes as suggested above because the metric does not calculate the similarities between the methods of the class. Therefore, we introduced

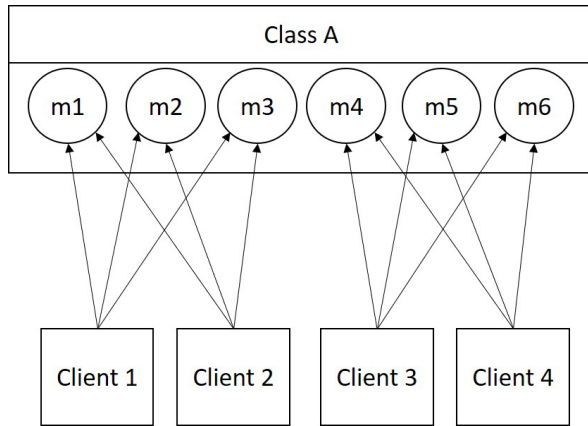


FIGURE 6. Class A and its clients.

in this article the cohesion metric $CCSM$ which can automatically support the Extract Class refactoring because the metric calculates the similarity between each pair of methods in the class.

C. THE PROPOSED APPROACH OF THE EXTRACT CLASS REFACTORING BASED ON $CCSM$

The proposed approach takes as an input a class to be refactored and automatically suggests a set of classes that can be extracted from the original class. The suggested classes should have higher cohesion than the original class. The flow chart in Fig. 7 shows the main steps of our approach. The first step is to parse the code of the class and extract its methods. Next, we extract the clients of each method in the class by parsing the code of the other classes in the system to identify client classes according to Definition 4.

Algorithm 1 is applied next to classify the methods of the class into disjoint sets based the proposed metric $CCSM$. The algorithm takes as an input the set of methods of the class and a refactoring threshold value and returns as an output a list (L) of disjoint sets of methods each of which represents a candidate class that may be extracted from the original class. The algorithm creates the disjoint sets one by one. The input threshold value is used to determine if a method to be classified is added to the last created set or added into a new set. If the refactoring threshold value is high and the client similarities between the methods of the class are generally low, each method will probably be added into a different set, which means each extracted class will have only one method. On the other hand, if the threshold value is low and the client similarities between the methods are relatively high, all the methods can be suggested to be placed into a single class, which means no class will be extracted from the input class. To mitigate this issue, we can compute the $CCSM$ for each pair of methods (in the input class) as a separate class that has only two methods and then choose the median of the non-zero results as a refactoring threshold value.

Algorithm 2 is applied next to merge small sets of methods with other sets to avoid extracting classes with a small number of methods. The algorithm takes as an input the list of sets

Algorithm 1 An Algorithm for Classifying the Methods of the Class

Input: 1) $M(A)$: the set of methods of the class to be refactored (class A). 2) *Threshold*: a refactoring threshold value.

Output: L : a list of disjoint sets of methods

Initialisation:

```

1:  $L = []$ ;
2: while  $|M(A)| > 1$  do
3:   find the two methods  $m_i, m_j \in M(A)$  such that the
   value of  $CCSM(\{m_i, m_j\})$  is the largest compared to the
   value of  $CCSM$  for any other two methods in  $M(A)$ ;
4:   add  $m_i, m_j$  to a new set  $D$ ;
5:   if  $CCSM(D) < Threshold$  then
6:     break;
7:   else
8:     remove  $m_i, m_j$  from  $M(A)$ ;
9:     while  $|M(A)| > 0$  do
10:      find the method  $m_k \in M(A)$  such that the value of
       $CCSM(D + \{m_k\})$  is the largest compared to the
      value of  $CCSM(D + \{m_y\})$  for any method  $m_y \in$ 
       $M(A) - \{m_k\}$ 
11:      if  $CCSM(D + \{m_k\}) \geq Threshold$  then
12:        add  $m_k$  to  $D$ ;
13:        remove  $m_k$  from  $M(A)$ ;
14:      else
15:        break;
16:      end if
17:    end while
18:    add  $D$  to  $L$ ;
19:  end if
20: end while
21: while  $|M(A)| > 0$  do
22:   add each remaining method  $m_r$  in  $M(A)$  to a new set
   and add the set to  $L$ ;
23:   remove  $m_r$  from  $M(A)$ ;
24: end while
25: return  $L$ 

```

L resulting from Algorithm 1 and the minimum number of methods that each extracted class can have and returns as an output the list L after merging the small sets in list. Algorithm 2 merges any set D (in the input list) that has a number of methods less than the input minimum number of methods with the set X in the input list such that $X \neq D$ and the value of $CCSM(D + X)$ is the largest compared to the value of $CCSM(D + Y)$ for any Y in the input list such that $Y \neq D$ and $Y \neq X$.

The final step in our approach is to suggest the extraction of classes based on the output of Algorithm 2 where each set in the output of the algorithm represents a candidate class that can be extracted from the original class. The attributes of the class are not considered in our approach. However, they can be automatically distributed among the extracted classes suggested by our approach based the use of the attributes by

Algorithm 2 An Algorithm for Merging Small Candidate Classes

Input: 1) L : the output of Algorithm 1 which may include small sets of methods. 2) $minNumMethods$: the minimum number of methods that each extracted class can have.

Output: L after merging small sets

Initialisation:

- 1: **for** D in L **do**
- 2: **if** $|D| < minNumMethods$ **then**
- 3: find X in L such that $X \neq D$ and the value of $CCSM(D + X)$ is the largest compared to the value of $CCSM(D + Y)$ for any Y in L such that $Y \neq D$ and $Y \neq X$;
- 4: add the elements of D to X ;
- 5: remove D from L ;
- 6: **end if**
- 7: **end for**
- 8: **return** L

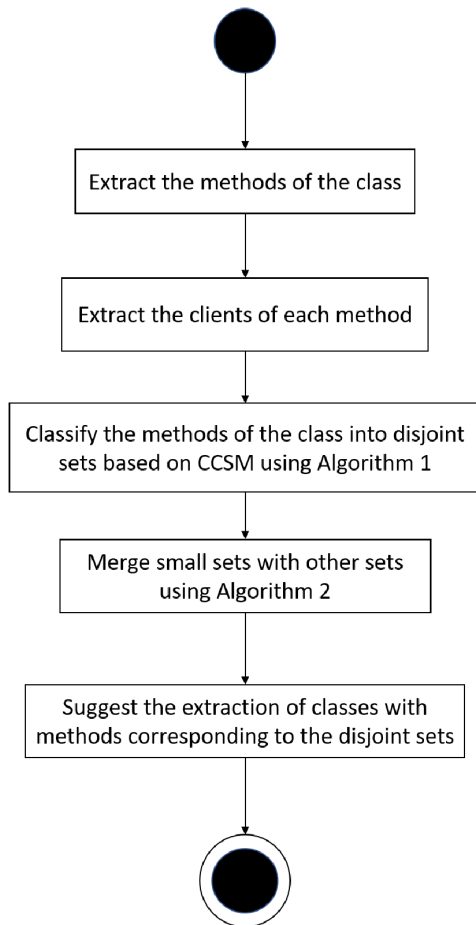


FIGURE 7. The process of the proposed approach for extract class refactoring.

the methods in the extracted classes. Each attribute can be added to the extracted class that has the largest number of methods that use the attribute. Our approach automatically suggests the set of classes that can be extracted from a given

TABLE 9. The $CCSM$ values for each pair of methods in the class c_1 .

	m2	m3	m4	m5	m6	m7	m8	m9	m10
m1	1	0.667	0	0	0	0	0	0	0
m2		0.667	0	0	0	0	0	0	0
m3			0	0	0	0	0	0	0.25
m4				0.667	0.667	0	0	0	0
m5					1	0	0	0	0
m6						0	0	0	0
m7							1	0.667	0
m8								0.667	0
m9									0

TABLE 10. The considered Blobs from GanttProject 1.10.2.

Class Name	#Methods	Clients
GanttProject	90	40
GanttGraphicArea	43	7
GanttTree	48	12
ResourceLoadGraphicArea	29	4
TaskImpl	46	41

class. A software engineer may evaluate the suggested classes and approve them or make some changes to them by moving the methods and attributes between the suggested classes.

D. AN EXAMPLE OF APPLICATION

To better understand the proposed approach, we present an example to show how the approach can be applied. Suppose we want to perform the Extract Class refactoring on the class c_1 that has the following set of methods:

$$M(c_1) = \{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9, m_{10}\}.$$

Suppose each method has a set of clients as follows:

$$\begin{aligned} Clients_M(m_1) &= \{c_2, c_3\}, \\ Clients_M(m_2) &= \{c_2, c_3\}, \\ Clients_M(m_3) &= \{c_2, c_3, c_4\}, \\ Clients_M(m_4) &= \{c_5, c_6\}, \\ Clients_M(m_5) &= \{c_5, c_6, c_7\}, \\ Clients_M(m_6) &= \{c_5, c_6, c_7\}, \\ Clients_M(m_7) &= \{c_8, c_9\}, \\ Clients_M(m_8) &= \{c_8, c_9\}, \\ Clients_M(m_9) &= \{c_8, c_9, c_{10}\}, \\ Clients_M(m_{10}) &= \{c_4, c_{11}\}. \end{aligned}$$

In order to select an appropriate refactoring *Threshold* value, we calculate the $CCSM$ for each pair of methods (in the class c_1) as a separate class that has two methods. The results are given in Table 9. The median of the non-zero values reported in Table 9 is chosen as a refactoring *Threshold* value, which is 0.667.

Algorithm 1 is applied next to classify the methods of c_1 into disjoint sets of methods. The output of Algorithm 1 is the following list:

$$L = [\{m_1, m_2, m_3\}, \{m_4, m_5, m_6\}, \{m_7, m_8, m_9\}, \{m_{10}\}].$$

The list L contains 4 sets. Each of the first three sets has three methods, while the last set in list has only one method. Algorithm 2 is applied on the list L to avoid extracting classes

TABLE 11. The results of refactoring the considered Blobs using the proposed approach.

Class Name	Pre-refactoring		Refactoring Threshold	Post-refactoring	
	#Methods	CCSM		#Methods	CCSM
GanttProject	90	0.002	0.143	15	0.057
				8	0.054
				11	0.055
				16	0.059
				22	0.058
				9	0.06
				5	0.067
				4	0.079
				GanttGraphicArea	43
33	0.392				
3	0.667				
3	0.667				
2	1.0				
GanttTree	48	0.032	0.5	31	0.368
				6	0.333
				5	0.26
				3	0.583
				3	0.667
ResourceLoadGraphicArea	29	0.279	1.0	23	0.474
				4	0.5
				2	1.0
TaskImpl	46	0.035	0.182	9	0.204
				16	0.147
				14	0.134
				4	0.25
				3	0.333

that have a small number of methods. We set the value of *minNumMethods* to 2, which specifies the minimum number of methods that each extracted class can have. The output of Algorithm 2 is the following:

$$L = [\{m_1, m_2, m_3, m_{10}\}, \{m_4, m_5, m_6\}, \{m_7, m_8, m_9\}].$$

Algorithm 2 merged the set $\{m_{10}\}$ with the set $\{m_1, m_2, m_3\}$ to avoid extracting a class that has only the method m_{10} . Each set in the list L after applying Algorithm 2 represents a candidate class that can be extracted from the original class c_1 .

The values of *CCSM* for the extracted classes $\{m_1, m_2, m_3, m_{10}\}$, $\{m_4, m_5, m_6\}$, and $\{m_7, m_8, m_9\}$ are 0.292, 0.778, and 0.667, respectively, whereas the value of *CCSM* for the original class c_1 is 0.044. The values of *CCSM* improved significantly for the extracted classes compared to the original class.

E. EXTRACT CLASS REFACTORING CASE STUDY

We present a case study to empirically evaluate the proposed approach of the Extract Class refactoring. For this purpose, we apply the proposed approach on real Blobs selected from GanttProject version 1.10.2. GanttProject is an open source tool implemented in Java that is used for scheduling and managing projects. We evaluate our approach by comparing the cohesion of the original classes (Blobs) with cohesion the extracted classes suggested by our approach. We use the proposed metric *CCSM* to measure the cohesion of the considered classes. We expect extracted classes to have higher *CCSM* values than the original classes.

Table 10 reports the names of the selected classes from GanttProject and the number of methods (excluding constructors) and clients of each class. We select these classes because

they were identified as Blobs in previous research [12], [41] and because each of them has at least two clients.

Table 11 reports the results obtained by applying the proposed Extract Class refactoring approach on the considered Blobs. The refactoring threshold value for each Blob is set to the median of the non-zero values resulting from computing the *CCSM* for each pair of methods in the Blob. The minimum number of methods that an extracted class can have is set to two. The results in Table 11 include the number of methods and *CCSM* values of the original classes and extracted classes. They indicate that the proposed approach is potentially useful as the cohesion of the extracted classes suggested by the approach are higher than the cohesion of the original classes.

VII. CONCLUSION AND FUTURE WORK

The paper introduced a client-based cohesion metric that supports the Extract Class refactoring. The metric was theoretically evaluated using the cohesion properties and it was empirically evaluated by comparing its measurements with the measurements of other cohesion metrics based on classes extracted from three open source systems. In addition, the paper proposed a refactoring approach that employed the newly introduced cohesion metric to automatically identify Extract Class refactoring opportunities. A case study based on real classes selected from an open source system showed the potential usefulness of the proposed refactoring approach.

A future study can extend the refactoring approach presented in this article by using the proposed client-based cohesion metric in combination with other well-known internal view-based cohesion metrics to better support the Extract Class refactoring. In addition, the use of client-based

cohesion to predict fault-prone classes is left open for future research.

REFERENCES

- [1] M. Alzahrani, S. Alqithami, and A. Melton, "Using client-based class cohesion metrics to predict class maintainability," in *Proc. IEEE 43rd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jul. 2019, pp. 72–80.
- [2] J. Al Dallal, "Object-oriented class maintainability prediction using internal quality attributes," *Inf. Softw. Technol.*, vol. 55, no. 11, pp. 2028–2048, Nov. 2013.
- [3] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Upper Saddle River, NJ, USA: Prentice-Hall, 2006.
- [4] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for cohesion measurement in object-oriented systems," *Empirical Softw. Eng.*, vol. 3, no. 1, pp. 65–117, 1998.
- [5] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, 1974.
- [6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
- [7] J. M. Bieman and B.-K. Kang, "Cohesion and reuse in an object-oriented system," *ACM SIGSOFT Softw. Eng. Notes*, vol. 20, no. 51, pp. 259–262, Aug. 1995.
- [8] C. Bonja and E. Kidanmariam, "Metrics for class cohesion and similarity between methods," in *Proc. 44th Annu. Southeast Regional Conf. (ACM-SE)*, 2006, pp. 91–95.
- [9] A. Marcus, D. Poshvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 287–300, Mar. 2008.
- [10] J. Al Dallal and L. C. Briand, "A precise method-method interaction-based cohesion metric for object-oriented classes," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 2, pp. 1–34, Mar. 2012.
- [11] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2009, pp. 93–101.
- [12] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *J. Syst. Softw.*, vol. 84, no. 3, pp. 397–414, Mar. 2011.
- [13] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: An improved method and its evaluation," *Empirical Softw. Eng.*, vol. 19, no. 6, pp. 1617–1664, Dec. 2014.
- [14] M. Alzahrani and A. Melton, "Defining and validating a client-based cohesion metric for object-oriented classes," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jul. 2017, pp. 91–96.
- [15] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp. 68–86, Jan. 1996.
- [16] S. Mäkelä and V. Leppänen, "Client-based cohesion metrics for java programs," *Sci. Comput. Program.*, vol. 74, nos. 5–6, pp. 355–378, Mar. 2009.
- [17] *Jhotdraw*. Accessed: Jun. 29, 2020. [Online]. Available: <https://github.com/wumpz/jhotdraw/releases>
- [18] *Argouml*. Accessed: Jun. 29, 2020. [Online]. Available: <https://github.com/argouml-tigris-org/argouml/releases>
- [19] *Apache Xerces2 Java*. Accessed: Jun. 29, 2020. [Online]. Available: <https://github.com/apache/xerces2-j/releases>
- [20] J. Al Dallal, "Measuring the discriminative power of object-oriented class cohesion metrics," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 788–804, Nov. 2011.
- [21] J. Al Dallal, "Fault prediction and the discriminative powers of connectivity-based object-oriented class cohesion metrics," *Inf. Softw. Technol.*, vol. 54, no. 4, pp. 396–416, Apr. 2012.
- [22] A. Marcus and D. Poshvanyk, "The conceptual cohesion of classes," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2005, pp. 133–142.
- [23] *Javaparser*. Accessed: Jun. 29, 2020. [Online]. Available: <https://javaparser.org/>
- [24] *The R Project for Statistical Computing*. Accessed: Jun. 29, 2020. [Online]. Available: <https://www.r-project.org/>
- [25] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.
- [26] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: Identification and application of extract class refactorings," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, 2011, pp. 1037–1039.
- [27] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *Proc. 5th Eur. Conf. Softw. Maintenance Reeng.*, 2001, pp. 30–38.
- [28] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th IEEE Int. Conf. Softw. Maintenance*, Sep. 2004, pp. 350–359.
- [29] K. J. Stewart, D. P. Darcy, and S. L. Daniel, "Opportunities and challenges applying functional data analysis to the study of open source software evolution," *Stat. Sci.*, vol. 21, no. 2, pp. 167–178, May 2006.
- [30] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring—improving coupling and cohesion of existing code," in *Proc. 11th Work. Conf. Reverse Eng.*, 2004, pp. 144–151.
- [31] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 264–282, Mar. 2011.
- [32] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proc. 8th Annu. Conf. Genetic Evol. Comput. (GECCO)*, 2006, pp. 1909–1916.
- [33] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui, "Automatic package coupling and cycle minimization," in *Proc. 16th Work. Conf. Reverse Eng.*, 2009, pp. 103–112.
- [34] M. O'Keefe and M. O. Cinneide, "Search-based software maintenance," in *Proc. Softw. Maintenance Reeng. (CSMR)*, 2006, p. 10.
- [35] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "A two-step technique for extract class refactoring," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2010, pp. 151–154.
- [36] G. Pappalardo and E. Tramontana, "Suggesting extract class refactoring opportunities by measuring strength of method interactions," in *Proc. 20th Asia-Pacific Softw. Eng. Conf. (APSEC)*, vol. 2, Dec. 2013, pp. 105–110.
- [37] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in eclipse: The ARIES project," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 1419–1422.
- [38] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, and Y.-G. Gueheneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–5.
- [39] M. Alzahrani, "Using clients to support extract class refactoring," in *Proc. 18th Int. Conf. Softw. Eng. Res. Pract.*, Jul. 2020.
- [40] M. Alzahrani and S. Alqithami, "An external client-based approach for the extract class refactoring: A theoretical model and an empirical approach," *Appl. Sci.*, vol. 10, no. 17, p. 6038, Aug. 2020.
- [41] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A Bayesian approach for the detection of code and design smells," in *Proc. 9th Int. Conf. Qual. Softw.*, Aug. 2009, pp. 305–314.



MUSAAD ALZHRANI received the B.Sc. degree from King Abdulaziz University, Jeddah, Saudi Arabia, in 2008, and the M.Sc. and Ph.D. degrees from Kent State University, Kent, OH, USA, in 2013 and 2017, respectively, all in computer science. He is currently an Assistant Professor with the Faculty of Computer Science and Information Technology, Albaha University, Al Bahah, Saudi Arabia. His research interests include software engineering, software metrics and qualities, software maintenance, and machine learning.

• • •