Journal of Systems Architecture 57 (2011) 452–462

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Dynamic cohesion measures for object-oriented software

Varun Gupta *, Jitender Kumar Chhabra

Department of Computer Engineering, National Institute of Technology, Kurukshetra, Kurukshetra 136 119, India

ARTICLE INFO

Article history:
Received 15 October 2009
Received in revised form 18 April 2010
Accepted 19 May 2010
Available online 26 May 2010

Keywords:
Cohesion
Dynamic metrics
Dynamic analysis
Software engineering
Object-oriented software systems

ABSTRACT

Most of the object-oriented cohesion metrics proposed in the literature define static cohesion at class level. Measurement of object-level dynamic cohesion however gives better insight into the behavioural aspects of the system. In this paper, dynamic cohesion metrics are introduced which provide scope of cohesion measurement up to object level and take into account important and widely used object-oriented features such as inheritance, polymorphism and dynamic binding during measurement. A theoretical framework is introduced before defining the measures and a theoretic validation of the proposed measures is carried out to make them more meaningful. A dynamic analyser tool is developed using aspect-oriented programming (AOP) to perform dynamic analysis of Java applications for the purpose of collecting run-time data for computation of the proposed dynamic cohesion measures. Further, an experiment is carried out for the proposed dynamic cohesion metrics using 20 Java programs and this study shows that the proposed dynamic cohesion metrics are more accurate and useful in comparison to the existing cohesion metrics. Moreover, the proposed dynamic cohesion metrics are validated empirically using source code APIs of Java Development Kit (JDK) and the proposed metrics are found to be better indicators of change-proneness of classes than the existing cohesion metrics.

1. Introduction

Object-oriented software is a collection of many classes and each class is a collection of attributes and methods. Software can be said to be of good quality if its classes have maximum cohesion and minimum coupling. Cohesion of a class is an internal software attribute representing the degree to which its elements are bound together. A large number of measures have been proposed to quantify this concept [7,9,11–14,19,20,28,33,37,38,40,41,43]. Most of the class cohesion measures proposed in literature are static in nature. Only a few attempts have been made to measure cohesion of a class at run-time [17,30,31]. However, these run-time cohesion measures are direct extensions of the existing static cohesion measures. In this paper, an attempt has been made to define the dynamic cohesion measures for objects as well as classes from the scratch and the proposed dynamic cohesion measures take into consideration the features of object-orientation like inheritance, polymorphism and dynamic binding. The inclusion of effect of these features in computation of cohesion makes them different and more accurate than the existing static cohesion measures. The dynamic cohesion metrics proposed in this paper are based on the executable code from which dynamic behaviour of applications is obtained. The method to measure object-level dynamic cohesion is to instrument the source code to log all occurrences of interactions among object-members while the application is being executed.

The main difference between static and dynamic cohesion metrics is the scope of measurement at which cohesion is being measured. The scope of static cohesion measurement is always the whole class. On the other hand, scope of dynamic cohesion measurement can even be specific to a single object belonging to a class at run-time. In case of dynamic metrics, cohesion is first measured at object level and then cohesion of corresponding class is obtained by aggregating cohesion values of all objects belonging to that class. Moreover, static cohesion metrics attempt to predict the potential interactions that would take place at run-time, whereas dynamic cohesion metrics measure what is actually happening at run-time rather than predicting.

With ever increasing use of object-oriented software in industry, it has been observed that inheritance and polymorphism are used more frequently to improve internal reuse in a system and facilitate maintenance [3]. The actual target of polymorphic method invocations can only be determined at run-time as per the inherited members of the class. Thus, it is not feasible to obtain precise static measures which take inheritance and polymorphism into account [2]. Dynamic cohesion measures are likely to be more accurate than static cohesion metrics for object-oriented programs involving inheritance and polymorphism.

The development of a well-defined cohesion measure consists of two steps: first, a well-defined theoretical framework should

* Corresponding author.
E-mail addresses: varun3dec@yahoo.com (V. Gupta), jitenderchhabra@rediffmail.com (J.K. Chhabra).

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.05.008

be constructed that characterizes the elements and relationships among the elements of an object or a class; second, a cohesion measure (satisfying well-defined theoretical properties) should be defined. This paper measures the dynamic cohesion on basis of four types of relationship among the elements of an object: the write dependence of attributes on methods, the read dependence of methods on attributes, the call dependence between methods and the reference dependence between attributes.

The remainder of this paper is organized as follows. Section 2 discusses the related work and Section 3 constructs a well-defined theoretical framework, proposes the definitions of dynamic cohesion metrics and performs theoretic validation of the proposed measures. Section 4 introduces a dynamic analyser tool for computation of proposed dynamic metrics using aspect-oriented approach and Section 5 presents a case study for the demonstration of computation of dynamic cohesion measures. Section 6 carries out an empirical study using 20 Java programs which assesses the proposed dynamic cohesion measures and compares them with the existing cohesion metrics. Section 7 validates the proposed measures empirically using source code APIs of JDK and Section 8 presents conclusions and future work directions.

2. Related work

A variety of cohesion metrics have been proposed and used in past empirical studies [7,9,11–14,19,20,28,33,37,38,40,41,43]. However, most of these measures are defined at class level. Only a few attempts have been made to define run-time cohesion metrics [17,30,31]. Gupta et al. [17] re-defined module cohesion metrics, SFC (Strong Functional Cohesion) and WFC (Weak Functional Cohesion) originally proposed by Bieman and Ott [7,37,38]. Gupta et al. [17] proposed program execution based module cohesion metrics based on the dynamic slicing of the program. They used dynamic slices of outputs to measure module cohesion. They stated that module cohesion metrics based on static slicing approach have got some inadequacies in cohesion measurement. The static measures significantly overestimate the levels of cohesion present in the software. Their approach addressed the drawbacks of static cohesion metrics by considering dynamic behaviour of programs and designing metrics based on dynamic slices obtained through program execution. The dynamic cohesion metrics are defined on the basis of common definitions, common use and common definition-use pairs in dynamic slices, which facilitate more precise cohesion measurement than the corresponding static metrics. The authors defined SFC as module cohesion obtained from common def-use pairs of each type common to the dynamic slices of all the output variables and WFC as module cohesion obtained from def-use pairs of each type found in dynamic slices of two or more output variables.

The proposal of Gupta et al. is an extension of Bieman's static cohesion and Mitchell and Power [30,31] defined dynamic cohesion metrics based on CK's LCOM (Lack of COhesion Metric) [4]. Mitchell and Power proposed two metrics Run-time Simple LCOM (R_{LCOM}) and Run-time Call-Weighted LCOM ($R^{\mathrm{W}}_{\mathrm{LCOM}}$) based on LCOM metric. First metric is a direct translation of the LCOM metric to take into account only those instance variables that are actually accessed at run-time. Second metric is an extension of the Run-time Simple LCOM, modified to take into account the total number of accesses made to an instance variable by a method of the class.

The dynamic cohesion metrics given by Gupta et al. [17] only deal with procedure-oriented program and the run-time cohesion metrics proposed by Mitchell et al. are just dynamic equivalent of the existing cohesion metrics such as LCOM given by Chidamber and kemerer [14]. To the best of our knowledge, this is the first attempt to define dynamic cohesion metrics from the beginning although a number of dynamic metrics have been proposed for the measurement of other software attributes such as coupling [42,2,3,21–23] and complexity [25,34,35]. Recently, empirical studies have indicated that static measures are insufficient for capturing dynamic aspects of object-oriented systems such as those related to inheritance and polymorphism [2,3]. Thus, it becomes important to define new metrics for the measurement of cohesion in object-oriented systems at run-time.

3. Dynamic cohesion measurement

In object-oriented systems, attributes and methods are the basic elements of an object or a class. A well-defined theoretical framework that formally defines these elements and depicts the relationships among the elements is the precondition of a well-defined cohesion measure. Here, a novel theoretical framework is proposed for characterizing elements and dependence relationships among elements of an object or class. This framework is used to describe relationships of four types: (i) write dependence relation between attributes and methods, (ii) read dependence relation between methods and attributes, (iii) call dependence relation among methods, and (iv) reference dependence relation among attributes.

The proposed measures take into account two types of access relationships between methods and attributes, i.e. read access relations and write access relations between methods and attributes. If a method having some logical error writes an attribute, then the value of attribute may also be incorrect. Thus, value of the attribute is dependent on the behaviour of the method during write access relationship between methods and attributes. Similarly, if a method reads an attribute that has incorrect value, then behaviour of the method may also be erroneous. Though, if method has a logical error, the value of attribute will not be affected by the method reading it. This fact states that the behaviour of method is dependent on the value of attribute during read access relationship between methods and attributes [1].

The proposed metrics account for inheritance and polymorphism present in object-oriented software. During dynamic cohesion measurement, we treat class (including inherited features) as a single semantic concept. Thus, set of attributes and set of methods of a class (formally defined in the next section) include set of inherited attributes and set of inherited methods, respectively. The concept of polymorphism is relevant only in method invocation type of connections. Since, cohesion is being measured for an object at run-time; polymorphic method invocations are accounted for automatically instead of static method invocations.

3.1. Definitions and terminology

Methods: $M(c)$ is the set of methods of a class c, which may be either inherited, or implemented in the class. Moreover, $M_{\mathrm{INH}}(c)$ is the set of methods inherited in class c and $M_{\mathrm{IMP}}(c)$ is the set of methods implemented (including overridden/re-defined methods) in class c. Further, $M_{\mathrm{DEF}}(c)$ is the set of methods defined in class c and $M_{\mathrm{REDEF}}(c)$ is the set of methods re-defined in class c. The following properties hold:

$$M_{\mathrm{IMP}}(c) = M_{\mathrm{DEF}}(c) \cup M_{\mathrm{REDEF}}(c)$$
$$M(c) = M_{\mathrm{INH}}(c) \cup M_{\mathrm{IMP}}(c)$$
$$M_{\mathrm{INH}}(c) \cap M_{\mathrm{IMP}}(c) = \emptyset$$

However, not all the methods of a class contribute to its cohesion [40]. There exist some special methods such as constructor, destructor, access methods and delegation methods intrinsically accessing only some of the attributes in the class [9,11,12,40]. A constructor is a type of method that initialises essential attributes

of the class and a destructor is a type of method that may only de-initialise crucial attributes of the class. An access method is a method that only reads or writes a particular attribute of the class. A delegation method is a method that only delegates a message to another object, especially to an attribute in the class, thus, generally has only one interaction with one attribute. These special methods may not essentially access all of the attributes. It has been widely accepted by a number of authors that these methods have no influence on the cohesion of a class [9,11,12,40]. Thus, these methods need to be excluded in the measurement of cohesion of a class. To describe this proposal better, special methods and normal methods are defined as follows:

$M_S(c)$ is the set of special methods in class c, which may be constructors, destructors, access methods or delegation methods in class c and $M_N(c)$ is the set of normal methods in class c, which are not special methods. The following properties also hold:

$$M(c) = M_S(c) \cup M_N(c)$$
$$M_S(c) \cap M_N(c) = \emptyset$$

Similarly, $M^R(o)$ is the set of methods used by an object o at run-time and $M^R_S(o)$ is the set of special methods used by object o and $M^R_N(o)$ is the set of normal methods used by object o and all above-defined properties also hold for methods used by an object at run-time.

Attributes: $A(c)$ represents set of all attributes whether implemented or inherited in a class c. Furthermore, $A_{INH}(c)$ and $A_{IMP}(c)$ are the set of attributes inherited and implemented respectively in class c. Also, $A(c) = A_{INH}(c) \cup A_{IMP}(c)$ and $A_{INH}(c) \cap A_{IMP}(c) = \emptyset$.

Similarly, $A^R(o)$ represents the set of attributes used by object o at run-time and all above-defined properties also hold for attributes used by the object at run-time.

Class: A class in an object-oriented system is the encapsulation of attributes and methods. However, if no relationship exists between members of a class, then there is no point of encapsulating them together. Thus, for the purpose of measurement of cohesion, a class must be considered as a set of its members and set of relationships between its members. A class, c in an object-oriented system can be defined as a set of its elements (attributes and methods) and relations between these elements, i.e. $c = \langle E, R\rangle$, where E represents the set of elements of a class, which may be attributes (instance variables) and methods (member functions), i.e. $E \in A(c) \cup M(c) \wedge c \in C$, where C represents the set of classes in an object-oriented system. The elements in a class may be implemented in the class itself or may be inherited from super classes. R represents set of relations among elements, E of the class.

Object: Objects are concrete representations of the classes and set of values of attributes of an object represent state of that object. Methods of an object define behaviour of the object at run-time. An object o is an instance of a class c created at run-time such that $o \in O(c) \wedge c \in C$ where $O(c)$ represents the set of objects of a class c created during execution of the program. An object o is defined in terms of its elements (attributes and methods) and relations existing at run-time between these elements, i.e. $o = \langle E^R, R^R\rangle$, where E^R represents the set of elements (attributes and methods) used by object o at run-time, i.e. $E^R \in A^R(o) \cup M^R(o)$ and R^R is a set of relations among elements, E^R at run-time.

Relations: The relation between a pair of elements e^R_i and e^R_j of an object at run-time is represented by $r^R_T\left(e^R_i, e^R_j\right)$ where T describes the type of relationship. The presence of a relation from element e^R_i to e^R_j at run-time is denoted by $r^R_T\left(e^R_i, e^R_j\right) = 1$. Moreover, this relation depicts that element e^R_i depends on e^R_j at run-time. The presence of two types of elements of an object (i.e. attributes and methods) causes four types of relations among object-elements. These types of relations between object-elements are described as follows:

Attribute–Method Write Relation $\left(r^R_W\left(e^R_i, e^R_j\right)\right)$: This type of dependence relation exists between attribute type element and method type element of an object. This relation exists when a method m of an object writes value of an attribute a of the object. In this sort of relation, attribute a depends on the method m due to the fact that value of attribute a is dependent on the behaviour of method m and this relation is termed as write dependency between attribute and method. This kind of relation between a pair of elements of an object, $o(o = \langle E^R, R^R\rangle$, where $\left(r^R_W\left(e^R_i, e^R_j\right) \in R^R \wedge e^R_i \in E^R \wedge e^R_j \in E^R\right)$ is represented as:

$$r^R_W\left(e^R_i, e^R_j\right) = 1 \wedge e^R_i \in A^R(o) \wedge e^R_j \in M^R(o) \wedge o \in O(c) \wedge c \in C$$

Method–Attribute Read Relation $\left(r^R_R\left(e^R_i, e^R_j\right)\right)$: This kind of dependence relation exists between a method and an attribute of an object when a method reads an attribute of the object at run-time. Due to this relation, method depends on the attribute since behaviour of the method is dependent on the value of the attribute being read and this type of relation is referred as read dependency between method and attribute. This kind of relation between a pair of elements of an object, $o(o = \langle E^R, R^R\rangle$, where $\left(r^R_R\left(e^R_i, e^R_j\right) \in R^R \wedge e^R_i \in E^R \wedge e^R_j \in E^R\right)$ is denoted as:

$$r^R_W\left(e^R_i, e^R_j\right) = 1 \wedge e^R_i \in M^R(o) \wedge e^R_j \in A^R(o) \wedge o \in O(c) \wedge c \in C$$

Method–Method Call Relation $\left(r^R_C\left(e^R_i, e^R_j\right)\right)$: This type of relation is present between a pair of method type elements of an object. This form of relation comes into picture when a method calls another method of the object o at run-time and this relation is known as call dependency between methods. In this relation calling method depends on the method being called. This type of relation accounts for polymorphism present in object-oriented programs as method being invoked by other method may be inherited one. This relation between a pair of elements of an object, $o(o = \langle E^R, R^R\rangle)$, where $\left(r^R_R\left(e^R_i, e^R_j\right) \in R^R \wedge e^R_i \in E^R \wedge e^R_j \in E^R\right)$ is represented as:

$$r^R_W\left(e^R_i, e^R_j\right) = 1 \wedge e^R_i \in M^R(o) \wedge e^R_j \in M^R(o) \wedge o \in O(c) \wedge c \in C$$

Attribute–Attribute Reference Relation $\left(r^R_{RF}\left(e^R_i, e^R_j\right)\right)$: This kind of relation exists between a pair of attribute type elements of an object at run-time. The reference of a pair of attributes together in a single method induces some dependency between them. An attribute a_i is said to be related to the other attribute a_j of an object o, if both are referred together in a method of an object and this relation is acknowledged as reference dependency between attributes. This particular type of relation is not directional in nature, since the relation between a pair of attributes is indirect in nature. Thus, $r^R_{RF}\left(e^R_i, e^R_j\right) = r^R_{RF}\left(e^R_j, e^R_i\right)$. This type of relation between a pair of elements of an object ($o = \langle E^R, R^R\rangle$, where $\left(r^R_{RF}\left(e^R_i, e^R_j\right) \in R^R \wedge e^R_i \in E^R \wedge e^R_j \in E^R\right)$ is represented as:

$$r^R_{RF}\left(e^R_i, e^R_j\right) = 1 \wedge e^R_i \in A^R(o) \wedge e^R_j \in A^R(o) \wedge o \in O(c) \wedge c \in C$$

Dependence Degrees of Relations: The above-defined four types of relations between members of an object or class have different degrees of dependency in terms of their importance towards measurement of cohesion. The dependence degrees of relations are characterized by weights shown in Table 1. Generally, the weightage of write dependency is greater than that of read dependency [40]. Thus, Attribute–Method Write Relation, $r^R_W\left(e^R_i, e^R_j\right)$ has got the more weightage than that of Method–Attribute Read Relation,

Table 1
Weightage of different relations.

Relation type	Weightage
Attribute–Method Write Relation $\left(r_W^R\left(e_i^R, e_j^R\right)\right)$	w_1
Method–Attribute Read Relation $\left(r_R^R\left(e_i^R, e_j^R\right)\right)$	w_2
Method–Method Call Relation $\left(r_C^R\left(e_i^R, e_j^R\right)\right)$	w_3
Attribute–Attribute Reference Relation $\left(r_{RF}^R\left(e_i^R, e_j^R\right)\right)$	w_4

$r_R^R\left(e_i^R, e_j^R\right)$, i.e. $w_1 > w_2$. The dependence relations between methods and attributes are more important than that of dependence relations among methods due to the fact that most of the cohesion measures are defined in terms of degree of tightness among methods and attributes [14,19,20,7,37,38]. Thus, Method–Attribute Read Relation, $r_R^R\left(e_i^R, e_j^R\right)$ has got more weightage than that of Method–Method Call Relation, $r_C^R\left(e_i^R, e_j^R\right)$, i.e. $w_2 > w_3$ and Attribute–Attribute Reference Relation, $r_{RF}^R\left(e_i^R, e_j^R\right)$ has got the lowest weightage due to the fact that this type of relation is indirect in nature, i.e. $w_1 > w_2 > w_3 > w_4$. The values of these weights can be assigned based on opinions of software engineering experts having sufficient (more than 10 years) analysis and design experience [27].

3.2. Definitions of measures

The mapping level of dynamic cohesion measurement can be either object or class. Object-level dynamic cohesion quantifies the extent of dependencies between the members of an object at run-time. As object is an instance of a class created at run-time during execution of a specific execution scenario x stimulated by input data or event. Dynamic cohesion of an object is obtained by averaging the measurements made for all scenarios of the application. Class-level dynamic cohesion aggregates the object-level cohesion values of all instances of a class created during all execution scenarios of the application. As defined above, there are four kinds of relations among attributes and methods to be considered for the measurement of dynamic cohesion of object or class. Thus, the dynamic cohesion of an object or class should be measured from the four facets.

The cohesion measurement of an object or class should consider only two types of elements: normal methods and attributes. In the following discussions, we assume that class c and so, the object o ($o \in O(c)$) has total number of attributes as m, i.e. $|A(c)| = m$ & $|A(o)| = m$ and total number of normal methods as n, i.e. $|M_N(c)| = n$ & $|M(o)| = n$.

(1) *Dynamic Cohesion due to Write dependency of Attributes on Methods* (DC_AM_X): This type of dynamic cohesion exists between attributes and methods of an object when a method of an object writes an attribute of the object during execution of a specific scenario x. This type of cohesion is due to the presence of relations of the type Attribute–Method Write Relation, $r_W^R\left(e_i^R, e_j^R\right)$ as defined above. This type of dynamic cohesion for an object o is defined as the ratio of actual number of distinct dependence relations of the type $r_W^R\left(e_i^R, e_j^R\right)$ between all attributes and all methods during execution of a specific scenario, i.e. $\sum_{i=1}^{m}\sum_{j=1}^{n} r_W^R\left(e_i^R, e_j^R\right)$ to the maximum possible number of relations of this type between them (i.e. $m \times n$). In case, if either number of attributes or number of methods of an object is zero, then this kind of cohesion is zero for that object. The dynamic cohesion of an object due to write dependency of attributes on methods execution of a specific scenario x is given as follows:

$$DC\_AM_X(o) = \begin{cases} 0 & m = 0 \text{ or } n = 0 \\ \dfrac{\sum_{i=1}^{m}\sum_{j=1}^{n} r_W^R\left(e_i^R, e_j^R\right)}{m \times n} & \text{where } e_i^R \in A(o) \wedge e_j^R \in M_N(o) \wedge o \in O(c) \end{cases}$$

(2) *Dynamic Cohesion due to Read dependency of Methods on Attributes* (DC_MA_X): This kind of cohesion exists between methods and attributes of an object when a method of an object reads an attribute during execution of a specific scenario x. This type of cohesion exists due to the relations of the type Method–Attribute Read Relation, $r_R^R\left(e_i^R, e_j^R\right)$ as defined above. This type of dynamic cohesion for an object o is defined as the ratio of actual number of distinct dependence relations of the type $r_R^R\left(e_i^R, e_j^R\right)$ between all methods and all attributes during execution of a specific scenario to the maximum possible number of relations of this type between them (i.e. $n \times m$). In case, if either number of methods or number of attributes are zero for an object then this type of cohesion is zero for that object. This kind of dynamic cohesion for an object o during execution of a specific scenario x is defined as follows:

$$DC\_MA_X(o) = \begin{cases} 0 & n = 0 \text{ or } m = 0 \\ \dfrac{\sum_{i=1}^{n}\sum_{j=1}^{m} r_R^R\left(e_i^R, e_j^R\right)}{n \times m} & \text{where } e_i^R \in M_N(o) \wedge e_j^R \in A(o) \wedge o \in O(c) \end{cases}$$

(3) *Dynamic Cohesion due to Call dependency between Methods* (DC_MM_X): This type of cohesion exists between a pair of methods of an object when a method m_i calls other method m_j of the object during program execution. This kind of cohesion is due to relations of the type, Method–Method Call Relation, $r_C^R\left(e_i^R, e_j^R\right)$ as already defined above. This kind of dynamic cohesion of an object o is defined as the ratio of actual count of distinct dependence relations of the type $r_C^R\left(e_i^R, e_j^R\right)$ between all ordered pairs of methods during execution of a specific scenario x to the maximum possible number of relations of this type between them (i.e. $n \times (n-1)$). In case, if number of methods of an object is zero then this type of cohesion is also zero for that object and if a single method exists for an object, then this type of cohesion is maximum, i.e. 1 for that object. This form of dynamic cohesion for an object o during execution of a specific scenario x is defined as follows:

$$DC\_MM_X(o) = \begin{cases} 0 & n = 0 \\ \dfrac{\sum_{i=1}^{n}\sum_{j=1 \wedge j \neq i}^{n} r_C^R\left(e_i^R, e_j^R\right)}{n \times (n-1)} & \text{where } e_i^R \in M_N(o) \wedge ej^R \in M_N(o) \wedge o \in O(c) \\ 1 & n = 1 \end{cases}$$

(4) *Dynamic Cohesion due to Reference dependency between attributes* (DC_AA_X): This category of cohesion exists between attributes of an object when these attributes are referred together in a method of the object during program execution. The reference of a pair of attributes together in a single method induces some dependency between them. This type of cohesion is due to the presence of relations of the type, Attribute–Attribute Reference Relation, $r_{RF}^R\left(e_i^R, e_j^R\right)$ as defined above. This kind of dynamic cohesion of an object o is obtained by dividing the actual number of distinct dependence relations of the type $r_{RF}^R\left(e_i^R, e_j^R\right)$ between all pairs of

attributes during execution of a specific scenario x to the maximum possible number of relations of this type between them (i.e. $n \times m \times (m-1)/2$). In case, if number of attributes of an object is zero then this type of cohesion is also zero for that object and if a single attribute exists for an object, then this type of cohesion is maximum, i.e. 1 for that object. This type of dynamic cohesion for an object o during execution of a specific scenario x is defined as follows:

$$DC\_AA_X(o) = \begin{cases} 0 & m = 0 \\ \dfrac{\sum_{i=1}^{m-1}\sum_{j=i+1}^{m} r_{RF}^{R}\left(e_i^R, e_j^R\right)}{n \times m \times (m-1)/2} & \text{where } e_i^R \\ & \in A(o) \wedge e_j^R \in A(o) \wedge o \in O(c) \\ 1 & m = 1 \end{cases}$$

Object-level dynamic cohesion: After measuring the above four aspects of dynamic cohesion for an object o separately during execution of a specific scenario x, overall dynamic cohesion of an object during execution of a specific scenario x is defined as the weighted summation of cohesion measures defined above. For this purpose, unequally weighted linear combination model [27] has been used since primitive metrics have unequal weights and there is no conflict situation in metric combination. According to this model, different weights are assigned to different primitive metrics to combine them into a single metric. The weights for different cohesion measures have been assigned as per the weightage given to their respective types of relations as given above in Table 1. The Dynamic Object Cohesion for an object o is defined as:

$$DOC_X(o) = \frac{w_1 * DC\_AM_X(o) + w_2 * DC\_MA_X(o) + w_3 * DC\_MM_X(o) + w_4 * DC\_AA_X(o)}{w_1 + w_2 + w_3 + w_4}$$

The values of the weights (w_1, w_2, w_3 and w_4) can be decided on the basis of opinions of software engineering experts having sufficient analysis and design experience [27]. An alternative approach for selection of values of weights could be to use the four base metrics in predictive models and estimate coefficients of those predictive models, which could be used as weights. This alternative approach could be explored in future work.

Dynamic cohesion of an object i in an application scope is the average of the dynamic cohesion values measured during all execution scenarios of the application for the object.

$$DOC(o_i) = \frac{\sum_{i=1}^{|X|} DOC_X(o_i)}{|X|}$$

where X is the set of scenarios in an application.

Class-level dynamic cohesion: Dynamic Class Cohesion is defined as the average of the values of Dynamic Object Cohesion for all objects of a class created during all execution scenarios of the application, i.e.

$$DCC(c) = \frac{\sum_{i=1}^{k} DOC(o_i)}{k} \quad \text{where } o_i \in O(c)$$

where k is the number of objects of class c created during all execution scenarios of the application.

One problem with measuring the dynamic cohesion of a class with application scope is to determine when the trace can stop such that the obtained cohesion values represent the complete application. One solution to this problem is to execute a new scenario and look if any object of the class has been covered by the scenario or there is any change in the value of dynamic cohesion of the class as a result of running the execution scenario. Unfortunately, this is not adequate to ensure that all scenarios in the application have been executed at least once. Thus, the person doing the trace needs to have a quite good knowledge of the application to ensure collection of accurate dynamic cohesion data for the computation of dynamic cohesion of a class (DCC).

Standard deviation is the most widely used measure of dispersion of values and is defined as the square-root of the average of squares of deviations, when such deviations for the values of individual items in a set of values are obtained from the arithmetic average [26]. Since Dynamic Class Cohesion (DCC) is the arithmetic average of the values of Dynamic Object Cohesion (DOC) of its objects, standard deviation can be used to measure dispersion in values of objects' cohesions. In case value of standard deviation of dynamic class cohesion of a class is beyond an acceptable limit, then the calculated dynamic class cohesion (DCC) of a class can be put under scanner and dynamic cohesion values of individual objects can be examined. The difference of dynamic cohesion of each object (DOC) from dynamic class cohesion (DCC) is calculated and objects having large differences can be identified and may be scrutinized further. The values of acceptable limits may be decided by software engineers as per the context of the programming environment. The standard deviation for a dynamic cohesion of a class c is calculated as:

$$\sigma_{DCC}(c) = \sqrt{\frac{\sum_{i=1}^{k}(DOC(o_i) - DCC(c))^2}{k}}, \quad \text{where } o_i \in O(c)$$

where k is the number of objects of class c created at run-time, $DOC(o_i)$ is the dynamic cohesion of object i and $DCC(c)$ is the dynamic cohesion of class c as defined above.

3.3. Theoretic validation

The four cohesion properties defined by Briand characterize cohesion in a reasonably intuitive and rigorous manner [8]. A well-defined cohesion measure should have the following four properties. These properties provide a guideline to develop a good cohesion measure.

Property 1 (*Non-negativity and Normalization*). Normalization of a cohesion measure makes it possible to carry out meaningful comparisons between the cohesion values of classes or objects having different number of elements, since they all belong to the same interval [8]. As per the definitions of the above-defined measures, the cohesion of an object or a class c lies within a specified range, i.e. $DOC \in [0,1]$ and $DCC \in [0,1]$. Thus, Property 1 holds for the proposed cohesion measures.

Property 2 (*Null value*). This property states that if there is no relationship among the elements of an object or class, then the cohesion of that object or class should be null. According to the definitions of the above-defined measures, if there is no relation exists between the elements of an object at run-time, then the values of all four types of cohesions for an object o, i.e. $DC\_AM(o)$, $DC\_MA(o)$, $DC\_MM(o)$ and $DC\_AA(o)$ will be zero and as a result cohesion of object o, $DOC(o)$ will also be zero, since $DOC(o)$ is the weighted summation of these measures. Moreover, the cohesion of a class c will also be null if cohesion values of all objects of the class are null. Thus, the proposed measures satisfy Property 2.

Property 3 (*Monotonicity*). This property requires that by addition of relationships among elements of an object or a class should not decrease its cohesion.

Let object, $o = \langle E^R, R^R \rangle$, where R^R represents the set of relations among set of elements, E^R of an object o at run-time. Let a relationship is added to o to form a new object $o' = \langle E^R, R^{R'} \rangle$, which is identical to o except that $R^R \subset R^{R'}$. Then, as per the above given definition of the measure, dynamic cohesion value of new object will only increase or will remain the same but will never decrease.

For objects, $o = \langle E^R, R^R \rangle$ and $o' = \langle E^R, R^{R'} \rangle$, if $R^R \subset R^{R'}$ then $DOC(o) \leqslant DOC(o')$. Similarly, the property holds for a class also. Thus, the proposed measures satisfy this property very well.

Property 4 (*Merging of objects or classes).* This property states that the cohesion of an object or a class obtained by putting together two unrelated objects or classes is not greater than the maximum cohesion of the two original objects or classes. If two unrelated objects o_1 and o_2 are merged to form a new object o_3 then the cohesion of o_3 is no larger than the maximum cohesion of o_1 and o_2 or if two unrelated classes c_1 and c_2 are merged to form a new class c_3, then cohesion of c_3 is no larger than the maximum cohesion of c_1 and c_2.

For, $o_1 = \langle E_1^R, R_1^R \rangle$ and $o_2 = \langle E_2^R, R_2^R \rangle$ where $R_1^R \cap R_2^R = \phi$ and $c_1 = \langle E_1, R_1 \rangle$ and $c_2 = \langle E_2, R_2 \rangle$ where $R_1 \cap R_2 = \phi$.

Two unrelated objects or classes have been combined to form a new object or class; there is a proportionate increase in number of relations as well as in number of elements. As per the definition of cohesion measures which measure cohesion in terms of ratio of actual number of relations existing at run-time divided by the maximum possible number of relations among elements. There is no net increase in the value of cohesion measure since numerator values as well as denominator values have increased together. Thus, the cohesion value of the combined object or class cannot be more than the maximum of the two unrelated objects or classes, i.e.

$$Max\{DOC(o_1), DOC(o_2)\} \geqslant DOC(o_3) \text{ or } Max\{DCC(c_1), DCC(c_2)\} \geqslant DCC(c_3)$$

Therefore, Property 4 also holds for the proposed cohesion measures.

4. Dynamic analysis for computation of dynamic cohesion metrics

Dynamic analysis of programs is a prerequisite for the calculation of dynamic metrics. Dynamic analysis of an application involves the collection of run-time data from the run-time profiles or from dynamic models of the software system. Dynamic analysis is more precise specially in handling object-oriented features like inheritance, polymorphism. Dynamic analysis of software can be performed in many ways: using profilers [32], from dynamic models [42] and using aspect-oriented programming [18]. Some other less popular techniques for dynamic analysis like AST rewriting based, pre-processor based, method-wrappers based and hybrid approaches also exist. After examination of all these methods, it is found that aspect-oriented approach provides a more desirable support for dynamic analysis of programs as compared to rest of the methods [18]. Moreover, aspect-oriented approach is easier to implement and at the same time, an efficient technique for dynamic analysis without any side effects.

4.1. AOP approach

Aspect-oriented programming (AOP) is a way of modularising crosscutting concerns much like object-oriented programming is a way of modularising common concerns. Aspect-oriented approach can be used for dynamic analysis of applications written in many languages by integrating the code with appropriate implementation of aspect-oriented programming. A number of implementations of AOP are available such as Aspectj for Java language, AspectC [44] for C language, AspectC++ [45] for C++ language, Aspect.NET [46] for C# and VB.NET languages. AspectJ [47] adds to Java a few new constructs: join-points, pointcuts, advices, and aspects. A join-point is just a name for an existing Java concept. It is any well-defined point in the program flow. A pointcut picks out certain join-points and values at these points. Advice is a piece of code that is executed when a join-point is reached and aspects are like Java classes, but may also include pointcuts, advices as its members. These aspects can be used as the interceptive code for the dynamic analysis of an application as they are written independently and merged with the target Java application using a weaving tool provided by the language environment. These aspects can be used for performing dynamic analysis of an application by tracing the application at run-time. Tracing can be seen as a concern that crosscuts the entire system and this concern is to be handled by encapsulating it into an aspect. Moreover, tracing is absolutely independent of what the system is doing. Thus, tracing can be performed without having any side effects on the basic functionality of the system and it can be plugged and unplugged whenever required [18].

4.2. Dynamic analyser implementation

We used AspectJ [47] to develop the interceptive code for Java applications as an aspect, which is an independent programming

```
public aspect DynamicAnalyser{

    //Pointcuts defined
    pointcut traceMethods() : (execution (* *.*(..)) ) ...
    pointcut traceAttribs() : ( (get(* *) || set(* *) ) ...
    pointcut traceRead() : get(* *) && withincode(* *.*(..)) ...
    pointcut traceWrite() : set(* *) && withincode(* *.*(..)) ...
    pointcut traceMethodsCalled() : call(* *.*(..)) ...
    // Advices defined for capturing pointcuts
    before(): traceMethods(){
            Signature sig=thisJoinPointStaticPart.getSignature();
            ...
    }
    after(): traceAttribs(){
            Signature sig=thisJoinPointStaticPart.getSignature();
            ...
    }
    after(): traceRead() {
    ...
    }
    after(): traceWrite() {
    ...
    }
     before(): traceMethodsCalled(){
    ...
    }
    // methods storing data collected at run-time into files
    void writeToFile_traceRead(Signature sig)
    {
    ...
    }
    void writeToFile_traceWrite(Signature sig)
    {
    ...
    }
    void writeToFile_traceReadWrite(Signature sig)
    {
    ...
    }
    void writeToFile_traceMethodsCalled(Signature sig)
    {
    ...
    }
} //aspect
```

Fig. 1. Main features of the dynamic analyser tool.

unit and can be merged with the target Java program without disturbing the basic functionality of the target program. Fig. 1 provides the key features of the dynamic analyser tool implemented in AspectJ. This analyser code is an independent unit and does not interfere with the target program and only interacts with the target program at run-time without altering the behaviour of the target program.

5. Case study

Consider a program written in Java [36] shown in Fig. 2. This program consists of a class Stack. The class Stack consists of two attributes and five methods: a constructor and four normal methods push, pop, topOfStack and isStackEmpty.

According to above given definitions, class Stack can be represented as Stack = $\langle E, R \rangle$, where

$E(Stack)$ = {stck[ ], tos, Stack(int), push(int), pop( ), isStackEmpty( ), topOfStack( )}
$A(Stack)$ = {stck[ ], tos}

```
Class Stack {
int stck[];
int tos;

Stack(int size)
{
        stck=new int [size];
        tos=-1;
}

void push(int item)
{
        if (tos = = stck.length-1)
        System.out.println("Stack is full");
        else
        stck[++tos]=item;
} //push method

int pop()
 {
        if(isStackEmpty()) {
        System.out.println("Stack Underflow");
        return 0; }
        else
        return stck[tos--];
} //pop method
int isStackEmpty() { return tos = = -1; }
int topOfStack() { return stck[tos-1];  }

public static void main(String str[])
{
Stack s1=new Stack(5);

// push numbers into stack
for(int i=0; i<5; i++) s1.push(i);

//pop numbers from stack
for(int i=0; i<5; i++) System.out.println(s1.pop());
} //main method
} //Stack class
```

Fig. 2. Example of a Java program.

$M(Stack)$ = {Stack(int), push(int), pop( ), isStackEmpty( ), topOfStack( )}
$M_N(Stack)$ = {push(int), pop( ), isStackEmpty( ), topOfStack( )}
$M_S(Stack)$ = {Stack(int)}

Here, $m = 2$ and $n = 4$

For the purpose of this case study, values of weights for different relations as given in Table 1 are taken as follows:

$$w_1 = 4, \quad w_2 = 3, \quad w_3 = 2 \quad \text{and} \quad w_4 = 1$$

On the execution of the above program along with the dynamic analyzer program, numbers of different types of relations recorded are as follows:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} r_W^R\left(e_i^R, e_j^R\right) = 3, \quad \sum_{i=1}^{n}\sum_{j=1}^{m} r_R^R\left(e_i^R, e_j^R\right) = 7,$$

$$\sum_{i=1}^{n}\sum_{j=1\wedge j\neq i}^{n} r_C^R\left(e_i^R, e_j^R\right) = 1 \quad \text{and} \quad \sum_{i=1}^{m-1}\sum_{j=i+1}^{m} r_{RF}^R\left(e_i^R, e_j^R\right) = 1$$

As per the definitions of different cohesion measures, their dynamic cohesion values calculated are as follows:

$DC\_AM\ (s1) = 3/2 * 4 = 0.375$
$DC\_MA\ (s1) = 7/4 * 2 = 0.875$
$DC\_MM\ (s1) = 1/4 * 3 = 0.833$
$DC\_AA\ (s1) = 1/4 * 2 * 1/2 = 0.25$

Thus, dynamic cohesion values of object *s1* and class *Stack* are calculated as follows: $DOC\ (s1) = (4 * 0.375 + 3 * 0.875 + 2 * 0.833 + 1 * 0.25)/10 = 0.60$. $DCC\ (Stack) = 0.60$.

6. Experimental study

We conducted an experimental study for the proposed metrics using 20 classes developed in Java. These classes were taken from text-book [36] and web. We used the dynamic analyser tool to calculate the above proposed dynamic cohesion measures for these 20 classes and results obtained are shown in Table 2. For the purpose of this experiment, values of weights for different relations are taken as: $w_1 = 4$, $w_2 = 3$, $w_3 = 2$ and $w_4 = 1$.

In Fig. 3, values of DCC metric for 20 classes are shown graphically. In this figure, Class Sr. Nos. as given in Table 2 are taken on

Table 2
Dynamic cohesion measurement.

Sr. No.	Class	DCC
1	Circle	0.40
2	Rectangle	0.20
3	Complex	0.90
4	Account	0.70
5	Manager	0.67
6	FixedStack	0.60
7	Balance	0.40
8	Triangle	0.40
9	RecTest	0.30
10	Box	0.40
11	Clicker	0.24
12	IceCream	0.60
13	BangBean	0.20
14	Atomicity	0.70
15	ChopStick	0.55
16	CompType	0.12
17	GeneratorsTest	0.30
18	InterestBearingAccount	0.45
19	Car	0.10
20	Employee	0.18

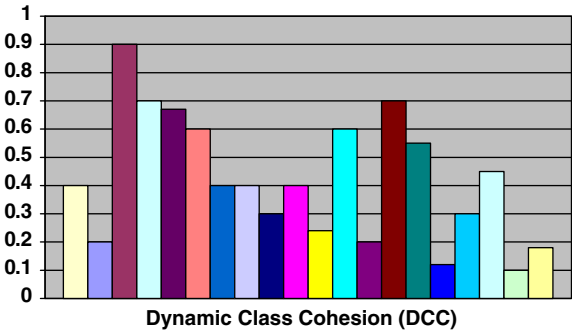

Fig. 3. DCC values for 20 classes taken for study.

the *X*-axis and their corresponding DCC values are plotted on the *Y*-axis.

Table 3 provides common descriptive statistics of the proposed metric distributions across all the classes used in the experimental study. This table shows that dynamic cohesion metric (DCC) values have a good variance in the data set.

6.1. Comparison between dynamic cohesion measures, DCC and R_{LCOM}

A correlation analysis using the standard Karl Pearson Product–Moment method is performed to compare the proposed dynamic cohesion metric (DCC) with the already existing dynamic cohesion metric, R_{LCOM} given by Mitchell et al. [30,31]. In this analysis, correlation between the dynamic cohesion metric, R_{LCOM} and widely used static cohesion measure, Lack of Cohesion in Methods measure (LCOM) given by Chidamber and Kemerer [14] is calculated and then, correlation between the proposed dynamic cohesion metric, DCC and LCOM is found to determine usefulness of the proposed measures. Correlation is a common research method used for empirical validation of the metrics. For instance, Li and Henry [29] examined correlations between various metrics in order to determine their usefulness. Kabaili et al. [24] used this statistical method to validate cohesion metrics as changeability indicators and Mitchell et al. [32] used this method to examine the usefulness of the run-time coupling metrics.

Correlation is measured using the standard Karl Pearson Product–Moment correlation coefficient *r*, which measures the degree and direction of the linear relationship between the two variables [26]. Karl Pearson's coefficient is the most widely used method of measuring correlation. This method assumes that there is a linear relationship between two variables and one of the variables is independent while the other is dependent. Karl Pearson's coefficient of correlation [26] is given by:

$$r = \frac{\sum(X_i - \overline{X})(Y_i - \overline{Y})}{n * \sigma_x * \sigma_y}$$

where X_i is the *i*th value of *X* variable; Y_i the *i*th value of *Y* variable; $\overline{X}$ mean of *X*; $\overline{Y}$ the mean of *Y*; *n* the number of pairs of observations of *X* and *Y*; σ_x the standard deviation of *X*; and σ_y is the standard deviation of *Y*.

Table 3
Descriptive statistics of dynamic cohesion metric (DCC).

Statistical parameter	Dynamic cohesion metric (DCC)
Maximum value	0.90
Minimum value	0.10
Median	0.40
Mean	0.42
Standard deviation	0.18

The value of correlation coefficient (*r*) lies between −1 and +1 through 0, where 1 represents a perfect positive correlation between the variables; −1 denotes a perfect negative correlation; and 0 indicates that there is no linear relationship between the variables. The degree of the correlation is determined by the magnitude of the coefficient. Adjective ratings of correlation strength follow the definitions developed by Cohen [15]:

- <0.1 "Trivial"
- 0.1–0.3 "Minor"
- 0.3–0.5 "Moderate"
- 0.5–0.7 "Large"
- 0.7–0.9 "Very large"
- 0.9–1 "Almost perfect"

Any relationship between two variables should be assessed for its strength as well as its significance. The significance of the correlation results is assessed by the *p*-value. The *p*-value corresponds to the probability that the measured correlation could be due to purely random effects. The smaller the *p*-level, more significant is the relationship between the variables.

Table 4 presents values of cohesion measures, LCOM, R_{LCOM} and DCC for all 20 classes undertaken for empirical study purpose. As stated above, the Karl Pearson Product–Moment correlation method is used to calculate the correlation between the different pairs of cohesion measures. The computed value of the correlation coefficient (*r*) between R_{LCOM} and LCOM comes out be 0.95 at the significance level of 0.05. Also, the value of the correlation coefficient (*r*) between the proposed metric, DCC and LCOM metric comes out to be 0.02 at the significance level of 0.05. Table 5 depicts the corresponding values of correlation coefficient (*r*) between different measures along with the significance level (*p*-value) at which the correlation coefficient is obtained.

The high value of correlation coefficient (*r*) between R_{LCOM} and LCOM suggests that R_{LCOM} is quite similar to the static measure, LCOM and is not able to capture much dynamic information during measurement. The value of correlation coefficient between the proposed metric, DCC and LCOM metric is less than 0.1 which suggests only trivial correlation exists between the proposed dynamic cohesion metric, DCC and the static metric LCOM. This analysis proves that the proposed dynamic cohesion metric, DCC is different from LCOM metric and captures dynamic information in a better way. Moreover, Table 4 shows that a large number of classes

Table 4
Values of LCOM, R_{LCOM} and DCC for classes.

Sr. No.	Class	LCOM	R_{LCOM}	DCC
1	Circle	2	1	0.40
2	Rectangle	0	0	0.20
3	Complex	0	0	0.90
4	Account	0	0	0.70
5	Manager	4	2	0.67
6	FixedStack	0	0	0.60
7	Balance	0	0	0.40
8	Triangle	0	0	0.40
9	RecTest	0	0	0.30
10	Box	0	0	0.40
11	Clicker	3	1	0.24
12	IceCream	0	0	0.60
13	BangBean	0	0	0.20
14	Atomicity	0	0	0.70
15	ChopStick	0	0	0.55
16	CompType	1	1	0.12
17	GeneratorsTest	0	0	0.30
18	InterestBearingAccount	0	0	0.45
19	Car	0	0	0.10
20	Employee	0	0	0.18

Table 5
Correlation among various measures.

Measures	Correlation coefficient (r)	Significance level (p-value)
R_{LCOM} and LCOM	0.95	0.05
DCC and LCOM	0.02	0.05

under study get assigned a zero value for R_{LCOM} metric. This happens due to the classes having a single method or when number of method-pairs accessing no common attributes is less than the number of method-pairs accessing common attributes. In comparison to this, the proposed dynamic cohesion metric, DCC shows a large variation of values for different classes as shown in Table 4. Thus, enough variance in run-time cohesion is captured by the DCC metric that is not accounted by R_{LCOM} metric. In addition to the above, the proposed dynamic cohesion metric, DCC also addresses the issues of access methods, constructors and impact of inheritance on the cohesion measurement, whereas R_{LCOM} metric does not address such issues in its definition.

7. Empirical validation of the proposed measures

In this section, we would conduct an experimental study to assess empirically whether the proposed dynamic cohesion measures are a useful predictor of external quality attribute such as change-proneness. This would help us to validate the proposed dynamic cohesion metrics as software quality indicators.

7.1. Experiment goal

The goal of this study is to analyze experimentally whether the proposed dynamic cohesion metrics are useful for predicting change-proneness of classes or not. For this purpose, we used the Goal/Question/Metric (GQM) paradigm [5,6,10,39] which provides a template as well as guidelines to define measurement goals in a systematic manner. The measurement goal of the experimental study is defined as follows:

- Object of study: class
- Purpose: prediction
- Quality focus: change-proneness
- Viewpoint: software developer
- Environment: source code APIs of JDK

These five goal dimensions have a direct impact on the remaining steps of the experimental validation of the proposed measures [10]. The object of study helps in defining the hypotheses that may be relevant as they are directly related to the object of study. The purpose helps to determine the type and amount of data to be collected. The quality focus facilitates in determining the dependent attribute(s) against which the defined metric is going to be experimentally validated. The viewpoint assists to determine the point in time at which predictions should be carried out. The environment helps to determine the context in which the experimental study is being carried out.

7.2. Empirical hypotheses

An empirical hypothesis is a statement believed to be true about the relationship between one or more attributes of the object of study and the quality focus [10]. In this case, the hypotheses are about the relationship between dynamic cohesion of a class (object of study) and change-proneness of the class (quality focus). In this experiment, we use the number of changed lines as an indicator of change-proneness.

The analysis is based on the hypotheses:

H_0: $p = 0$ (Null hypothesis) – There is no significant correlation between the proposed dynamic cohesion metrics and change-proneness of classes.

H_1: $p \neq 0$ (Alternative hypothesis) – There is significant correlation between the proposed dynamic cohesion metrics and change-proneness of classes.

7.3. Experimental environment

In order to experimentally validate the proposed metrics, we conducted an experimental study with source code APIs included in JDK which is a software development package from Sun Microsystems that implements the basic set of tools needed to write, test, and debug Java applications and applets. This experimental study has been performed with 15 packages having a total number of 1183 classes. This type of similar study has been carried out by Woo et al. [41] to show that the revised cohesion measures considering the impact of write interactions between class members have stronger relations with change-proneness of classes than the original cohesion measures.

7.4. Analysis methodology

A statistical analysis is performed to correlate the proposed dynamic cohesion metrics with change-proneness. In this analysis, the number of changed lines for a class is used as an indicator of change-proneness. In the experimental study, correlation is measured using the standard Karl Pearson Product–Moment correlation coefficient [26] which has already been explained in detail in Section 6.1 above. The statistical significance level (p-value) for the statistical test has been taken as 0.05 which corresponds to the probability that the measured correlation could be due to purely random effects. The smaller the p-level, more significant is the relationship between the variables.

7.5. Analysis of experimental results

In the experiment, we have performed an empirical study of the proposed dynamic cohesion measures with change-proneness. This study tries to find the effectiveness of the proposed dynamic cohesion measures on predicting the change-proneness of classes in comparison to the static cohesion measures. For this purpose, we first set JDK 1.3.10 as the baseline, and then count the number of changed lines in each class compared with JDK1.4.12, then compute the correlation coefficient between cohesion values and the number of changed lines for each measure under study including the static as well as the proposed dynamic cohesion measures. We use the number of changed lines as an indicator of change-proneness. Table 6 shows the total number of classes, the number of changed lines and number of changed classes for all packages between JDK 1.3.10 and JDK 1.4.12 [41]. We consider 1183 classes in our experiment and 626 classes are changed among these classes.

As stated above, the Karl Pearson Product–Moment correlation method (explained in detail in Section 6.1) was used to quantify the correlation between the cohesion metrics (proposed as well as existing) values for classes and the number of changed lines in classes. The number of changed lines in classes between above given two versions of JDK is used as an indicator of change-proneness of classes. Table 7 summarizes the results of the correlation study. The table shows the computed correlation coefficients of the cohesion metrics (including the proposed metric, DCC) with the change-proneness along with the significance level (p-value) at which the correlation coefficient is obtained.

Table 6
No. of changed classes and changed lines.

Package name	Total no. of classes	No. of changed classes	No. of changed lines
java.applet	1	1	4
java.awt	225	133	1448
java.beans	24	9	74
java.io	65	21	329
java.lang	93	39	329
java.math	5	3	32
java.net	33	21	329
java.rmi	48	12	75
java.security	82	26	186
java.sql	11	7	25
java.txt	50	23	315
java.util	48	33	386
javax.accessibility	9	4	17
javax.naming	63	9	24
javax.swing	426	285	4086
Total	**1183**	**626**	**7729**

Table 7
Results of correlation study.

Metric	Correlation coefficient	Significance level (*p*-value)
$LCOM1_w$	0.534	0.05
$LCOM2_w$	0.531	0.05
TCC_w	−0.088	0.05
DCC	**−0.63**	**0.05**

Table 7 shows that the proposed metric, DCC has a strong negative correlation with change-proneness of classes, i.e. −0.63 at significance level of 0.05. This negative correlation makes sense since classes with stronger cohesion (high value of DCC) are expected to have less number of changes. The amount of correlation between DCC and change-proneness is much higher than that of other reported cohesion metrics in Table 7.

Woo et al. have already shown that the revised cohesion measures, $LCOM1_w$, $LCOM2_w$, and TCC_w, have stronger relation with the change-proneness of classes than LCOM1, LCOM2, and TCC in [41]. The results of the correlation study conducted in this paper show that the proposed measures have got much stronger relation with change-proneness than the cohesion measures such as $LCOM1_w$, $LCOM2_w$, and TCC_w, which suggests that the proposed measures may be better indicators than the existing measures for predicating change-proneness. In other words, if a class undergoes frequent changes between different versions of software, then that class will have low dynamic cohesion as measured by the proposed metrics and if a class has got high value of dynamic cohesion (represents a single abstraction), then the class is less prone to changes between different versions of the software. The results show that the proposed metrics measure cohesion of a class more accurately which may be due to the fact that the proposed metrics measure cohesion of a class at run-time and take into account all major types of relations between class-members during measurement.

Since, the proposed measures have got strong correlation with the change-proneness of classes. We reject the null hypothesis and accept the alternative hypothesis (as stated above). The strong correlation between the proposed dynamic cohesion metrics (DCC) and change-proneness signify that the proposed metrics are a good indicator of the external quality attributes such as change-proneness of classes. Change proneness has been used as an indicator of maintainability [29] as well it might also be useful to identify maintenance "hot-spots" and unstable classes in software [2]. Moreover, number of lines changed can also be used to measure reusability of a class in order to extend its functionality in a prescribed way. The more lines required, the lower the reusability. This appears to us to be a rough but reasonable measure of the effort that would be required for a programmer to adapt a class for use within a larger system [16]. Therefore, the proposed dynamic cohesion metrics might be useful indicators of the software quality attributes such as change-proneness, maintainability and reusability.

8. Conclusion and future work

This paper proposes new well-defined dynamic cohesion measures. In the definitions of the measures, the elements as well as the relationships among the elements in a class are precisely characterized and the special methods which do not contribute to the cohesiveness of objects and classes are excluded in the measurement. The proposed measures are well-defined cohesion measures which satisfy the four cohesion properties defined by Briand et al. [8] and in comparison with the existing cohesion measures, the proposed measures have the following advantages:

- The proposed dynamic cohesion measures are more accurate as they are defined at run-time and take into consideration the actual interactions taking place rather than the potential interactions which may or may not happen as is the case with static cohesion metrics.
- The proposed cohesion metrics take inheritance and polymorphism into consideration as the actual targets of polymorphic invocations can only be determined at run-time due to the presence of inherited members of a class.
- The scope of measurement of the proposed dynamic cohesion metrics can be specific to a single object or even a scenario. The limiting of scope of measurement might be useful in testing and impact analysis [2]. Whereas, other existing cohesion metrics are able to measure cohesion up to class level only.
- The proposed dynamic cohesion measures are better indicators of external software quality attributes such as change-proneness than the existing static cohesion metrics as proved by the experimental study.

In future work, relationships between the proposed dynamic cohesion metrics and other quality attributes such as fault-proneness could be explored. Moreover, in future work, an alternative approach for selection of values of weights assigned to different relations could be explored to use the four base metrics in predictive models and estimate coefficients of the predictive models, which could be used as weights.

References

[1] H. Aman, T. Yanaru, M. Nagamatsu, K. Miyamoto, A metric for class structure complexity focusing on relationships among class members, IEICE Transactions on Information and System E81-D (12) (1998) 1364–1373.
[2] E. Arisholm, Dynamic coupling measures for object-oriented software, in: Proc. 8th IEEE Symp. on Software Metrics (METRICS'02), Ottawa, Canada, 2002, pp. 33–42.
[3] E. Arisholm, L.C. Briand, A. Føyen, Dynamic coupling measurement for object-oriented software, IEEE Transactions on Software Engineering 30 (8) (2004) 491–506.
[4] J. Bansiya, L.H. Etzkorn, C.G. Davis, W. Li, A class cohesion metric for object oriented designs, Journal of Object-oriented Programming 11 (8) (1999) 47–52.
[5] V.R. Basili, D. Weiss, A Methodology for collecting valid software engineering data, IEEE Transactions of Software Engineering 10 (11) (1984) 728–738.
[6] V.R. Basili, D.H. Rombach, The Tame Project: towards improvement-oriented software environments, IEEE Transactions of Software Engineering 14 (6) (1988) 758–773.
[7] J. Bieman, B. Kang, Cohesion and reuse in an object-oriented system, in: Proc. ACM Symp. Software Reusability (SSR'95), 1995, pp. 259–262 (reprinted in ACM SIGSOFT Software Engineering Notes, 1995).

[8] L.C. Briand, S. Morasca, V.R. Basili, Property-based software engineering measurement, IEEE Transactions on Software Engineering 22 (1) (1996) 68–85.
[9] L.C. Briand, J.W. Daly, J. Wust, A unified framework for cohesion measurement in object-oriented systems, Empirical Software Engineering 3 (1) (1998) 65–117.
[10] L. Briand, S. Morasca, V. Basili, An operational process for goal-driven definition of measures, IEEE Transactions on Software Engineering 28 (12) (2002) 1106–1125.
[11] H.S. Chae, Y.R. Kwon, A cohesion measure for classes in object-oriented systems, in: Proc. Fifth International Software Metric Symposium (METRICS'98), Bethesda, MD, USA, IEEE Computer Society Press, 1998, pp. 158–166.
[12] H.S. Chae, Y.R. Kwon, D.H. Bae, A cohesion measure for object oriented classes, Software Practice and Experience 30 (12) (2000) 1405–1431.
[13] Z. Chen, Y. Zhou, B. Xu, J. Zhao, H. Yang, A novel approach to measuring class cohesion based on dependence analysis, in: Proc. International Conference on Software Maintenance, IEEE Computer Society Press, Montreal, Canada, 2002, pp. 377–384.
[14] S.R. Chidamber, C.F. Kemerer, A metrics suite for object-oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[15] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, second ed., Lawrence Erlbaum Publishing Co., N.J., Mahwah, 1988.
[16] G. Gui, Component reusability and cohesion measures in object-oriented systems, in: Proc. Information and Communication Technologies, 2006, pp. 2878–2882.
[17] N. Gupta, P. Rao, Program execution based module cohesion measurement, in: Proc. 16th International Conference on Automated on Software Engineering (ASE '01), San Diego, USA, 2001.
[18] V. Gupta, J.K. Chhabra, Measurement of dynamic metrics using dynamic analysis of programs, in: Proc. WSEAS International Conference on Applied Computing Conference, Istanbul, Turkey, 2008, pp. 81–86.
[19] B. Henderson-Sellers, Software Metrics, Prentice Hall, Hemel Hempstaed, UK, 1996.
[20] M. Hitz, B. Montazeri, Measuring coupling and cohesion in object oriented systems, in: Proc. International Symposium on Applied Corporate Computing, Monterrey, Mexico, 1995, pp. 25–27.
[21] Y. Hassoun, R. Johnson, S. Counsell, A dynamic runtime coupling metric for meta-level architectures, in: Proc. Eur. Conf. On Software Maintenance and Reengineering (CSMR'04), IEEE Computer Society Press, 2004, pp. 339–346.
[22] Y. Hassoun, R. Johnson, S. Counsell, Empirical Validation of a Dynamic Coupling Metric, Technical Report BBKCS-04-03, School of Computer Science and Information Systems, Birkbeck College, University of London, UK, March 2004.
[23] Y. Hassoun, S. Counsell, R. Johnson, Dynamic coupling metric-proof of concept, IEE Proceedings Software 152 (6) (2005).
[24] H. Kabaili, R. Keller, F. Lustman, Cohesion as changeability indicator in object-oriented systems, in: Proc. IEEE Conference on Software Maintenance and Reengineering (CSRM), 2001, pp. 39–46.
[25] T.M. Khoshgoftaar, J.C. Munson, D.L. Lanning, Dynamic system complexity, in: Proc. Software Metrics Symposium, Baltimore, MD, USA, 1993, pp. 129–140.
[26] C.R. Kothari, Research Methodology: Methods & Techniques, New Age International Publishers, New Delhi, 2007.
[27] S.-T. Lai, A software metric combination model for software reuse, in: Proc. Asia Pacific Software Engineering Conference, Taipei, 1998, pp. 70–77.
[28] Y.S. Lee, B.S. Liang, Measuring the coupling and cohesion of an object-oriented program based on information flow, in: Proc. International Conference on Software Quality, Maribor, Slovenia, 1995, pp. 81–90.
[29] W. Li, S. Henry, Object oriented metrics that predict maintainability, Journal of Systems and Software 23 (2) (1993) 111–122.
[30] A. Mitchell, J.F. Power, Run-Time Cohesion Metrics for the Analysis of Java Programs, Technical Report Series No. NUIM-CS-TR-2003-08, National University of Ireland, Maynooth Co. Kildare, Ireland, 2003.
[31] A. Mitchell, J.F. Power, Run-time cohesion metrics: an empirical investigation, in: Proc. the International Conference on Software Engineering Research and Practice, 2004, pp. 532–537.
[32] A. Mitchell, J.F. Power, An empirical investigation into the dimensions of run time coupling in Java programs, in: Proc. third International Conference on Principles and Programming in Java (PPPJ'04), Las Vegas, Nevada, 2004, pp. 9–14.
[33] S. Moser, V.B. Misic, Measuring class coupling and cohesion: a formal meta-model approach, in: Proc. Asia Pacific Software Engineering Conference and International Computer Science Conference, IEEE Computer Society Press, Hong Kong, 1997, pp. 31–40.
[34] J.C. Munson, T.M. Khoshgoftaar, Measuring dynamic program complexity, IEEE Software 9 (6) (1992) 48–55.
[35] J.C. Munson, T.M. Khoshgoftaar, Software metrics for reliability assessment, in: Michael Lyu (Ed.), Handbook of Software Reliability Engineering, McGraw-Hill, 1996, pp. 493–529.
[36] P. Naughton, H. Schildt, Java 2: The Complete Reference, third revised ed., McGraw-Hill, 1999.
[37] L.M. Ott, J.M. Bieman, B.K. Kang, Developing measures of class cohesion for object-oriented software, in: Proc. Seventh Annual Oregon Workshop on Software Metrics, Oregon, Portland, 1995.
[38] L.M. Ott, J.M. Bieman, Program slices as an abstraction for cohesion measurement, Journal of Information and Software Technology 40 (11–12) (1998) 691–699.
[39] R. Van Solingen, The goal/question/metric approach, Encyclopedia of Software Engineering—2 Volume Set, 2002, pp. 578–583.
[40] J. Wang, Y. Zhou, L. Wen, Y. Chen, H. Lu, B. Xu, DMC: a more precise cohesion measure for classes, Information and Software Technology 47 (3) (2005) 167–180.
[41] G. Woo, H.S. Chae, J.F. Cui, J.-H. Ji, Revising cohesion measures by considering the impact of write interactions between class members, Information and Software Technology 51 (2) (2009) 405–417.
[42] S. Yacoub, H. Ammar, T. Robinson, Dynamic metrics for object-oriented designs, in: Proc. Int. Symp. on Software Metrics (Metrics'99), Boca Raton, FL, USA, 1999, pp. 50–61.
[43] Y. Zhou, B. Xu, J. Zhao, H. Yang, ICBMC: an improved cohesion measure for classes, in: Proc. International Conference on Software Maintenance, IEEE Computer Society Press, Montreal, Canada, 2002, pp. 44–53.
[44] AspectC, <http://www.cs.ubc.ca/labs/spl/projects/aspectc.html>.
[45] AspectC++, <http://www.aspectc.org/>.
[46] Aspect.net, <http://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=6801>.
[47] AspectJ, <http://www.eclipse.org/aspectj>.

Varun Gupta is pursuing his Ph.D. degree in the area of Software Engineering from Department of Computer Engineering, National Institute of Technology (Deemed University), Kurukshetra 136 119, India. He obtained his Bachelor of Technology degree in computer science and engineering from Guru Nanak Dev University, Amritsar in 1999 and Master of Engineering degree in software engineering from Thapar Institute of Engineering and Technology, Patiala (Deemed University) in 2003. He worked as a lecturer in Department of Computer Science and Engineering, RIMT Institute of Engineering and Technology for 4 years. Presently, he is working as Assistant Director in Directorate of Information Technology, PSEB, Patiala. His areas of interest include Software Engineering, Object Oriented Design and Development, and Data Mining.

Jitender Kumar Chhabra, Ph.D., is working as Assistant Professor in Department of Computer Engineering, National Institute of Technology (Deemed University), Kurukshetra 136 119, India. He received his B.Tech. in Computer Engineering as 2nd rank holder and M.Tech in Computer Engineering as Gold Medalist, both from Regional Engineering College, (now N.I.T.) Kurukshetra. He completed his PhD degree on Software Metrics from GGS Indraprastha University, Delhi, India. He is teaching in N.I.T. Kurukshetra since last 13 years. He has also worked in collaboration with companies like Hewellet-Packard and Tata Consultacy Services. He has published more than 50 research papers in various international and national journals and conferences including IEEE, Elsevier, ACM. He is reviewer of many reputed research journals like IEEE and Elsevier. He is adaptation author of Gottfried's Schaum-Series book on Programming with C from Tata McGraw Hill. His areas of interest include software engineering, data base system, data structure, programming techniques.