Empirical Software Engineering, 3, 65–117 (1998)
© 1998 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

A Unified Framework for Cohesion Measurement in Object-Oriented Systems

LIONEL C. BRIAND briand@iese.fhg.de
Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany

JOHN W. DALY
Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany

JÜRGEN WÜST wuest@iese.fhg.de
Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany

Abstract. The increasing importance being placed on software measurement has led to an increased amount of research developing new software measures. Given the importance of object-oriented development techniques, one specific area where this has occurred is cohesion measurement in object-oriented systems. However, despite a very interesting body of work, there is little understanding of the motivation and empirical hypotheses behind many of these new measures. It is often difficult to determine how such measures relate to one another and for which application they can be used. As a consequence, it is very difficult for practitioners and researchers to obtain a clear picture of the state-of-the-art in order to select or define cohesion measures for object-oriented systems.

This situation is addressed and clarified through several different activities. First, a standardized terminology and formalism for expressing measures is provided which ensures that all measures using it are expressed in a fully consistent and operational manner. Second, to provide a structured synthesis, a review of the existing approaches to measure cohesion in object-oriented systems takes place. Third, a unified framework, based on the issues discovered in the review, is provided and all existing measures are then classified according to this framework. Finally, a review of the empirical validation work concerning existing cohesion measures is provided.

This paper contributes to an increased understanding of the state-of-the-art: a mechanism is provided for comparing measures and their potential use, integrating existing measures which examine the same concepts in different ways, and facilitating more rigorous decision making regarding the definition of new measures and the selection of existing measures for a specific goal of measurement. In addition, our review of the state-of-the-art highlights several important issues: (i) many measures are not defined in a fully operational form, (ii) relatively few of them are based on explicit empirical models as recommended by measurement theory, and (iii) an even smaller number of measures have been empirically validated; thus, the usefulness of many measures has yet to be demonstrated.

Keywords: cohesion, object-oriented, measurement

1.0. Introduction

The market forces of today's software development industry have begun to place much more emphasis on software quality. This has led to an increasingly large body of work being performed in the area of software measurement, particularly for evaluating and predicting the quality of software. In turn, this has led to a large number of new measures being proposed for quality design principles such as cohesion. Modules of a high quality software design, among many other principles, should obey the principle of high cohesion. Stevens et al., who first introduced cohesion in the context of structured development techniques, define cohesion as a measure of the degree to which the elements of a module belong together. In a highly cohesive module, all elements are related to the performance of a single function. Such modules are hypothesized to be easier to develop, maintain, and reuse, and to be less fault-prone. Some empirical evidence exists to support this theory for systems developed by structured and object-based techniques; see, e.g., (Card, 1986; Card, 1985), and (Briand et al., 1994).

In object-oriented software, classes replace modules, with methods and attributes as their elements. In this context, cohesion is the degree to which the methods and attributes of a class belong together. Again, recent research has led to a large number of new cohesion measures for object-oriented systems being defined. However, because cohesion is a complex software attribute in object-oriented systems (e.g., there are several different mechanisms which are considered to contribute to the cohesion of a class), and there has been no attempt to provide a structured synthesis, our understanding of the state-of-the-art is poor. For example, because there is no standard terminology and formalism for expressing measures, many measures are not fully operationally defined, i.e., there is some ambiguity in their definitions. As a result, it is difficult to understand how different cohesion measures relate to one another. Moreover, it is also unclear what the potential uses of many existing measures are and how these different measures might be used in a complementary manner. The fact that there also exists little empirical validation of existing object-oriented cohesion measures means the usefulness of most measures is not supported.

To address and clarify our understanding of the state-of-the-art of cohesion measurement in object-oriented systems requires a comprehensive framework based on a standard terminology and formalism. This framework can then be used (i) to facilitate comparison of existing cohesion measures, (ii) to facilitate the evaluation and empirical validation of existing cohesion measures, and (iii) to support the definition of new cohesion measures and the selection of existing ones based on a particular goal of measurement. Analogous research for coupling measurement is described in Briand, Daly, and Wüst (1996). The coupling framework presented in that paper is considered to be complementary to the cohesion framework presented here.

The paper is organized as follows. Section 2.0 summarizes the current state of cohesion measurement in object-oriented system and provides detailed motivation for the need for the research performed in this paper. Section 3.0 introduces the notation and formalism required to conduct this research. Section 4.0 provides a comprehensive review and structured synthesis of existing object-oriented cohesion frameworks and measures. The results of this review are then used to define a new unified framework for cohesion measurement in object-oriented systems in Section 5.0. In Section 6.0, a review of empirical validation studies of cohesion measures takes place.

2.0. Motivation

Object-oriented measurement has become an increasingly popular research area. This is substantiated by the fact that recently proposed in the literature are (i) several different frameworks for coupling and cohesion and (ii) a large number of different measures for object-oriented attributes such as coupling, cohesion, and inheritance. While this is to be welcomed, there are several negative aspects to the mainly ad hoc manner in which object-oriented measures are being developed. As neither a standard terminology or formalism exists, many measures are expressed in an ambiguous manner which limits their use. This also makes it difficult to understand how different measures relate to one another. For example, there are many different decisions that have to be made when defining a cohesion measure—these decisions have to be made considering the measurement goal and by defining an empirical model based on clearly stated hypotheses. Unfortunately, for many measures proposed in the literature these decisions and hypothesis are not documented. It is therefore often unclear what the potential uses of existing measures are and how different cohesion measures could be used in a complementary manner to obtain a more detailed picture of the cohesion of classes in an object-oriented system. In short, our understanding of existing cohesion measures is not what it should be.

Several authors have introduced different approaches and proposed measures to characterize cohesion in object-oriented systems, e.g., Chidamber and Kemerer (1991, 1994), Hitz and Montazeri (1995), Bieman and Kang (1995), Henderson-Sellers (1996), Lee et al. (1995), Briand, Morasca, and Basili (1993, 1994). Eder et al. define a framework aimed at providing qualitative criteria for cohesion; they also assign relative strengths to different levels of cohesion they identify within this framework (Eder, Kappel, and Schrefl, 1994). However, neither this framework nor the different approaches used have characterized existing measures to the different dimensions of cohesion that have been identified. Therefore, the negative aspects highlighted above are still very prevalent ones. In our review of the literature, for example, we found 15 different measures[1] of object-oriented cohesion. Consequently, it is not difficult to imagine how confusing the overall picture actually is.

To make a serious attempt to improve our understanding of object-oriented cohesion measurement we have to integrate all existing approaches into a unique theoretical framework, based on a homogenous and comprehensive formalism. A review of existing measures has to be performed, and these measures have to be categorized according to the unified framework. This framework will then be a mechanism with which to compare measures and their potential use, integrate existing measures which examine the same concepts in a different manner, and allow more rigorous (and ease of) decision making regarding the definition of new measures and the selection of existing measures with respect to their utility. It should facilitate the evaluation and empirical validation of cohesion measures by ensuring that specific hypotheses are provided which link cohesion measures to external quality attributes. It should also facilitate identification of dimensions of cohesion which thus far have been overlooked, i.e., for which there are no measures defined. Finally, the framework must be able to integrate new cohesion measures as they are defined in the future. In that sense both the formalism and the framework must be extensible.

3.0. Terminology and Formalism

In the past, research within the area of software measurement has suffered from a lack of (i) standardized terminology and (ii) a formalism for defining measures in an unambiguous and fully operational manner (that is, a manner in which no additional interpretation is required on behalf of the user of the measure). As a consequence, development of consistent, understandable, and meaningful software quality predictors has been severely hampered.

For example, Churcher and Shepperd (1995a) and Hitz and Montazeri (1996) have identified ambiguities in members of the well referenced object-oriented suite of measures by Chidamber and Kemerer (1994). To remedy this situation it is necessary to reach a consensus on the terminology, define a formalism for expressing software measures, and, most importantly, to use this terminology and formalism. Of course, the level of detail and scope of the terminology and formalism required are subject to the goal to be achieved.

To rigorously and thoroughly perform a review and a structured synthesis of software cohesion measures we seek to define a terminology and formalism that is implementation independent and can be extended as necessary. This will allow all existing work to be expressed in a consistent, understandable, and meaningful manner and allow the measures reviewed to be expressed as operationally defined (additional interpretation of ambiguous measures is given when required). A disadvantage of this approach is that the reader must first be presented with the formalism before the review can begin in a meaningful fashion. Given the motivation for such an approach, however, it is argued that this is the only method to facilitate a rigorous and thorough review.

To prevent the reader having to read a complete terminology list we have provided a glossary in Appendix A, which includes definitions applicable to object-oriented systems and to measurement in general. This can be referenced as required. Where appropriate the terminology defined by Churcher and Shepperd (1995b) has been used.

To express the cohesion measures consistently and unambiguously the following formalism based on set theory is presented. Note that for the sake of brevity we assume that the reader is familiar with common object-oriented principles and needs no explanation of them. For those readers not so familiar, simple explanations by means of examples are provided in Appendix B.

System

Definition 1. System, classes, inheritance relationships

An object-oriented system consists of a set of classes, C. There can exist inheritance relationships between classes such that for each class $c \in C$ let

- $\mathit{Parents}(c) \subset C$ be the set of parent classes of class c.
- $\mathit{Children}(c) \subset C$ be the set of children classes of class c.
- $\mathit{Ancestors}(c) \subset C$ be the set of ancestor classes of class c.
- $\mathit{Descendents}(c) \subset C$ be the set of descendent classes of class c.

Methods

A class has a set of methods.

Definition 2. Methods of a class
For each class $c \in C$ let $M(c)$ be the set of methods of class c.

A method can be either virtual or non-virtual and either inherited, overridden, or newly defined, public or non-public, all of which have implications for measuring cohesion. It is therefore necessary to express the difference between these categories.

Definition 3. Declared and implemented methods
For each class $c \in C$, let

- $M_D(c) \subseteq M(c)$ be the set of methods *declared in* c, i.e., methods that c inherits but does not override or virtual methods of c
- $M_I(c) \subseteq M(c)$ be the set of methods *implemented in* c, i.e., methods that c inherits but overrides or non-virtual non-inherited methods of c

where $M(c) = M_D(c) \cup M_I(c)$ and $M_D(c) \cap M_I(c) = \emptyset$.

Definition 4. Inherited, overriding, and new methods
For each class $c \in C$ let

- $M_{INH}(c) \subseteq M(c)$ be the set of inherited methods of c.
- $M_{OVR}(c) \subseteq M(c)$ be the set of overriding methods of c.
- $M_{NEW}(c) \subseteq M(c)$ be the set of non-inherited, non-overriding methods of c.

Definition 5. Public and non-public methods
For each class $c \in C$, we define

- $M_{pub}(c) \subseteq M(c)$ the set of public methods of c, and
- $M_{npub}(c) \subseteq M(c)$ the set of non-public methods of c.

It is $M_{pub}(c) \cap M_{npub}(c) = \emptyset$ and $M_{pub}(c) \cup M_{npub}(c) = M(c)$. A public method can be accessed by any other method in the system. A non-public method can only be accessed by a certain subset of methods (for instance, in C++, a method declared *private* in class c can only be invoked by methods implemented in class c). Because the restrictions which apply to the access of non-public methods are language dependent, and because these restrictions are not important for the discussion of measures later on, we make no assumptions about the exact restrictions that apply to the access to non-public methods.

For notational convenience, we also define the set of all methods in the system, $M(C)$.

Definition 6. Set of all methods in the system

$M(C)$ is the set of all methods in the system and is represented as $M(C) = \bigcup_{c \in C} M(c)$.

Methods have a set of parameters which, as they also influence cohesion measurement, must be defined.

Definition 7. Parameters
For each method $m \in M(C)$ let $Par(m)$ be the set of parameters of method m.

Method Invocations

To measure the cohesion of a class, c, it is necessary to define the set of methods that $m \in M(c)$ invokes and the frequency of these invocations. Method invocations can be either static or dynamic; it is necessary to distinguish between these. Consequently, for each method $m \in M(C)$ the following sets are defined.

Definition 8. $SIM(m)$ the set of statically invoked methods of m
Let $c \in C$, $m \in M_I(c)$ and $m' \in M(C)$. Then $m' \in SIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' is invoked for an object of static type class d.

Definition 9. $NSI(m, m')$ the number of static invocations of m' by m
Let $c \in C$, $m \in M_I(c)$ and $m' \in SIM(m)$. $NSI(m, m')$ is the number of method invocations in m where m' is invoked for an object of static type class d and $m' \in M(d)$.

Definition 10. $PIM(m)$ the set of polymorphically invoked methods of m
Let $c \in C$, $m \in M_I(c)$ and $m' \in M(C)$. Then $m' \in PIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' may, because of polymorphism, be invoked for an object of dynamic type d.

Definition 11. $NPI(m, m')$ the number of polymorphic invocations of m' by m
Let $c \in C$, $m \in M_I(c)$ and $m' \in PIM(m)$. $NPI(m, m')$ is the number of method invocations in m where m' can be invoked for an object of dynamic type class d and $m' \in M(d)$.

As a result of polymorphism, one method invocation can contribute to the *NPI* count of several methods. Note that $\forall m \in M(C)$: $SIM(m) \subseteq PIM(m)$, and therefore $\forall m$, $m' \in M(C)$: $NSI(m, m') \leq NPI(m, m')$.

Indirect Method Invocations

For a method $m \in M(C)$, $SIM(m)$ and $PIM(m)$ are sets of methods directly invoked by m. We also need to define the sets of methods indirectly invoked by m. Method m indirectly invokes method m', if there are methods $m_1, m_2, \ldots, m_n$ such that m directly invokes m_1, m_1 directly invokes m_2, etc., and m_n directly invokes m'. This idea underlies the following definition.

Definition 12. Indirectly invoked methods
$\forall m \in M(C)$, let

$$SIM^*(m) = \{m' \mid m' \in M(C) \wedge \exists n \geq 1 \exists m_1, m_2, \ldots, m_n \in M(C)\colon m_1 = m \wedge m_n = m' \wedge \forall i, 1 < i \leq n\colon m_i \in SIM(m_{i-1})\}$$

$$PIM^*(m) = \{m' \mid m' \in M(C) \wedge \exists n \geq 1 \exists m_1, m_2, \ldots, m_n \in M(C)\colon m_1 = m \wedge m_n = m' \wedge \forall i, 1 < i \leq n\colon m_i \in PIM(m_{i-1})\}$$

Attributes

Classes have attributes which are either inherited or newly defined. Attributes are modelled using a similar formalism to that of methods.

Definition 13. Declared and implemented attributes
For each class $c \in C$ let $A(c)$ be the set of attributes of class c. $A(c) = A_D(c) \cup A_I(c)$ where

- $A_D(c)$ is the set of attributes declared in class c (i.e., inherited attributes).
- $A_I(c)$ is the set of attributes implemented in class c (i.e., non-inherited attributes).

Again, for notational convenience, we define the set of all attributes in the system, $A(C)$.

Definition 14. $A(C)$ the set of all attributes
$A(C)$ is the set of all attributes in the system and is represented as $A(C) = \bigcup_{c \in C} A(c)$.

Attribute References

Methods may reference attributes. It is sufficient to consider the static type of the object for which an attribute is referenced because attribute references are not determined dynamically. For the discussion of measures later, it must be possible to express for a method, m, the set of attributes referenced by the method:

Definition 15. $AR(m)$
For each $m \in M(C)$ let $AR(m)$ be the set of attributes referenced by method m.

Types

Attributes and parameters have types which all can contribute to cohesion. The programming language provides a basic set of built-in types; the user can define new class types as well as traditional types (e.g., records, enumerations). The traditional user-defined types may be defined locally in a class, or they may be of global scope.

Definition 16. Basic types, local and global user-defined types

- BT is the set of built-in types provided by the programming language (e.g., integer, real, character, string).
- UDT is the set of user-defined types of global scope (e.g., records, enumerations).
- For each class $c \in C$, $T(c)$ is the set of types defined within the scope of class c.

The type of an attribute or parameter either is a class, a built-in type or a user-defined type. Thus, the set $\boldsymbol{T}$ of available types in the system is defined as follows:

Definition 17. $\boldsymbol{T}$ the set of available types
The set $\boldsymbol{T}$ of available types in the system is $\boldsymbol{T} = BT \cup UDT \cup C \cup (\bigcup_{c \in C} T(c))$.

The next definition determines how the types of attributes and parameters will be denoted.

Definition 18. Types of attributes and parameters

For each attribute $a \in A(C)$ the type of attribute a is denoted by $T(a) \in \boldsymbol{T}$.

For each method $m \in M(C)$ and each parameter $v \in Par(m)$ the type of parameter v is denoted by $T(v) \in \boldsymbol{T}$.

No distinction is made between pointers, references, or arrays and the type they are derived from.

The notation and formalism defined, a mechanism is now available to express existing cohesion measures in a consistent and precise manner.

4.0. Survey of Cohesion Measurement Approaches and Measures

In this section we perform a comprehensive survey and critical review of existing approaches and measures for cohesion in object-oriented systems. The section is organized as follows. In Section 4.1, we present existing approaches and measures for cohesion. In Section 4.2, the approaches are discussed and compared. In Section 4.3, the cohesion measures derived from the various approaches are analyzed, which includes theoretical validation of these measures.

4.1. Existing Approaches to Measure Cohesion

Eder et al. (1994) propose a framework aimed at providing qualitative criteria for cohesion. Chidamber and Kemerer (1991, 1994), Hitz and Montazeri (1995), Bieman and Kang (1995), Henderson-Sellers (1996), Lee et al. (1995), and Briand et al. (1993, 1994) each propose different approaches to measure cohesion in object-oriented or object-based systems and define various cohesion measures accordingly. We discuss these frameworks and approaches in the following sections.

4.1.1. Framework by Eder et al. (1994)

Eder et al. (1994) proposed a framework aimed at providing comprehensive, qualitative criteria for cohesion in object-oriented systems. To that end, they adapted existing frameworks for cohesion in the procedural and object-based paradigm to the specifics of the object-oriented paradigm. They distinguish between three types of cohesion in an object-oriented system: method, class and inheritance cohesion. For each type, various degrees of cohesion exist. In the following, we will briefly explain the types of cohesion.

1. *Method cohesion.* Eder et al. apply Myers' classical definition of cohesion (Myers, 1978) to methods. Elements of a method are statements, local variables and attributes of the method's class. They define seven degrees of cohesion, based on the definition by Myers (1978). From weakest to strongest, the degrees of method cohesion are:

- *Coincidental*: The elements of a method have nothing in common besides being within the same method.
- *Logical*: Elements with similar functionality such as input/output handling are collected in one method.
- *Temporal*: The elements of a method have logical cohesion and are performed at the same time.
- *Procedural*: The elements of a method are connected by some control flow.
- *Communicational*: The elements of a method are connected by some control flow and operate on the same set of data.
- *Sequential*: The elements of a method have communicational cohesion and are connected by a sequential control flow.
- *Functional*: The elements of a method have sequential cohesion, and all elements contribute to a single task in the problem domain. Functional cohesion fully supports the principle of locality and thus minimizes maintenance efforts.

2. *Class cohesion.* Class cohesion addresses the relationships between the elements of a class. The elements of a class are its non-inherited methods and non-inherited attributes. Eder et al. use a categorization of cohesion for abstract data types by Embley and Woodfield (1987) and adapt it to object-oriented systems. There are five degrees of class cohesion.

From weakest to strongest, these are:

- *Separable*: The objects of a class represent multiple unrelated data abstractions. For instance, the cohesion of a class is separable, if the methods and attributes can be grouped into two sets such that any method of one set invokes no methods and references no attributes of the other set, and vice versa.

- *Multifaceted*: The objects of a class represent multiple related data abstractions. The relation is caused by at least one method of the class which uses all these data abstractions.

- *Non-delegated*: There exist attributes which do not describe the whole data abstraction represented by a class, but only a component of it.

- *Concealed*: There exist some useful data abstraction concealed in the data abstraction represented by the class. Consequently, the class includes some attributes and methods which might make another class. For instance, consider a class *Employee* having, amongst others, attributes *DayOfBirth*, *MonthOfBirth*, *YearOfBirth*, *DayOfHire*, *MonthOfHire*, and *YearOfHire*. These attributes describe a concealed data abstraction, "date." In this case, we would define a new class *Date* with attributes *Day*, *Month* and *Year*, and replace the date attributes in class *Employee* by two attributes *BirthDate* and *HireDate* of type *Date*.

- *Model*: The class represents a single, semantically meaningful concept.

3. *Inheritance cohesion.* Like class cohesion, inheritance cohesion addresses the relationships between elements of a class. However, inheritance cohesion takes all the methods and attributes of a class into account, i.e., inherited and non-inherited. Inheritance cohesion is strong if inheritance has been used for the purpose of defining specialized children classes. Inheritance cohesion is weak, if it has been used for the purpose of reusing code. The degrees of inheritance cohesion are the same as those for class cohesion.

Within this framework, an analysis of the semantics of a given method or class is required to determine its degree of method, class or inheritance cohesion. Such an analysis is likely to requires a good knowledge of the system's application domain, it is subjective, and it cannot be automated. If we use this framework to derive cohesion measures, the resulting measures will not be automatically collectable. The definitions of the degrees of cohesion in this framework should be used as guidelines to derive syntactically-based measures which are measuring approximations of these degrees of cohesion in a particular context.

4.1.2. Approach by Chidamber and Kemerer (1991, 1994)

Chidamber and Kemerer define cohesion measures which are theoretically based on the ontology of objects by Bunge (1977, 1979). Within this ontology, the *similarity* of things is defined as the set of properties the things have in common. Chidamber and Kemerer adapt this idea to define the cohesion of a class as the *degree of similarity* of its methods. The degree of similarity of a set M of methods is the number of attributes used in common by all methods in M, formally denoted by $\sigma(M) = |\bigcap_{m \in M} AR(m)|$. Chidamber and Kemerer argue that $\sigma(M_I(c))$ itself is not a suitable measure for the cohesion of a class c: if all but one method in c use the same set $A \subset A_I(c)$ of attributes, and the remaining method only uses attributes in $A_I(c) - A$, we have $\sigma(M_I(c)) = 0$, even though most methods of c are similar. Instead, Chidamber and Kemerer propose a cohesion measure LCOM defined as follows (Chidamber and Kemerer, 1991):

> Consider a Class C_1 with methods $M_1, M_2, \ldots, M_n$. Let $\{I_i\}$ = set of instance variables (note: attributes in our terminology) used by method M_i. There are n such sets $\{I_1\}, \ldots, \{I_n\}$.
> LCOM = The number of disjoint sets formed by the intersection of the n sets.

LCOM is an *inverse* cohesion measure. A high value of LCOM indicates low cohesion and vice versa. The above definition of LCOM has been interpreted in different ways by different authors. The interpretation by Hitz and Montazeri (1995) will be discussed in Section 4.1.3. Henderson-Sellers offers the following interpretation (Henderson-Sellers, 1996): $LCOM' = |\{I_i \cap I_j = \emptyset \mid \forall i, j, i \neq j\}|$, i.e., the number of pairs of methods in class c having no common attribute references. Using our formalism, we define LCOM' as follows (and refer to this definition as LCOM1):

Definition 19 (Measure LCOM1).

$$LCOM1(c) = |\{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) = \emptyset\}|$$

Note that this definition only considers methods *implemented* in class c, and that only references to attributes *implemented* in class c are counted. This is an additional interpretation of our own; the influence of inheritance on the cohesion of a class has not been addressed by Henderson-Sellers nor by Chidamber and Kemerer.

In Chidamber and Kemerer (1994), Chidamber and Kemerer give a new definition of LCOM:

> Consider a Class C_1 with methods $M_1, M_2, \ldots, M_n$. Let $\{I_i\}$ = set of instance variables used by method M_i. There are n such sets $\{I_1\}, \ldots, \{I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. If all n sets $\{I_1\}, \ldots, \{I_n\}$ are $\emptyset$ then let $P = \emptyset$.
>
> $$LCOM = \begin{cases} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{otherwise} \end{cases}$$

LCOM is the number of pairs of methods in a class having no common attribute references, $|P|$, minus the number of pairs of similar methods, $|Q|$. However, if $|P| < |Q|$, LCOM is set to zero. The definition of this version of LCOM is almost operational. Again, it is not stated whether inherited methods and attributes are included or not. Using our formalism, we define this measure as follows:

Definition 20 (measure LCOM2).

Let $P = \begin{cases} \emptyset, \text{ if } AR(m) = \emptyset \ \forall m \in M_1(c) \\ \{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) = \emptyset\}, \text{ else} \end{cases}$

Let $Q = \{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) \neq \emptyset\}$.

Then $LCOM2(c) = \begin{cases} |P| - |Q|, \text{ if } |P| > |Q| \\ 0, \text{ otherwise} \end{cases}$

4.1.3. Approach by Hitz and Montazeri (1995)

Hitz and Montazeri base their approach to measure cohesion on the work of Chidamber and Kemerer. They interpret the definition of LCOM in Chidamber and Kemerer (1991) as follows (Hitz and Montazeri, 1995):

> Let X denote a class, I_X the set of its instance variables, and M_X the set of its methods. Consider a simple, undirected graph $G_X(V, E)$ with $V = M_X$ and $E = \{(m, n) \in V \times V \mid \exists i \in I_X\colon (m \text{ accesses } i) \wedge (n \text{ accesses } i)\}$. LCOM is then defined as the number of connected components of G_X.

This definition is almost operational. It is not stated whether inherited methods and attributes are included or excluded in the sets I_X and M_X. We rewrite the definition of Hitz and Montazeri using our formalism:

Definition 21 (measure LCOM3).
Let $G_c = (V_c, E_c)$ be an undirected graph with vertices $V_c = M_I(c)$ and edges $E_c = \{\{m_1, m_2\} \mid m_1, m_2 \in V_c \wedge AR(m_1) \cap AR(m_2) A_I(c) \neq \emptyset\}$. *LCOM3(c)* is the number of connected components of G_c.

Again, our definition considers only methods and attributes implemented in class c.

Hitz and Montazeri identified a problem with *access methods* for LCOM3. An access method provides read or write access to an attribute of the class. Access methods typically reference only one attribute, namely the one they provide access to. If other methods of the class use the access methods, they may no longer need to directly reference any attributes at all. These methods are then isolated vertices in graph G_c. Thus, the presence of access methods artificially decreases the class cohesion as measured by LCOM3. There is no empirical justification for this artificial loss of cohesion. To remedy this problem, Hitz and Montazeri propose a second version of their LCOM measure. In this version, the definition of graph G_c is changed as follows: there is also an edge between vertices representing methods m_1 and m_2, if m_1 invokes m_2 or vice versa. We define this measure as follows.

Definition 22 (measure LCOM4).
Let $G_c = (V_c, E_c)$ be an undirected graph with vertices $V_c = M_I(c)$ and edges $E_c = \{\{m_1, m_2\} \mid m_1, m_2 \in V_c \wedge (AR(m_1) \cap AR(m_2) \cap A_I(c) \neq \emptyset \vee m_1 \in SIM(m_2) \vee m_2 \in SIM(m_1))\}$. *LCOM4(c)* is the number of connected components of G_c.

In the case where G_c consists of only one connected component ($LCOM4(c) = 1$), the number of edges $|E_c|$ ranges between $|V_c| - 1$ (minimum cohesion) and $|V_c| \cdot (|V_c| - 1)/2$ (maximum cohesion). Hitz and Montazeri define a measure Co ("connectivity") which further discriminates classes having $LCOM4(c) = 1$ by taking into account the number of edges of the connected component.

Definition 23 (measure Co).

$$Co(c) = 2 \cdot \frac{|E_c| - (|V_c| - 1)}{(|V_c| - 1) \cdot (|V_c| - 2)}, \text{ where } E_c \text{ and } V_c \text{ are defined as in Definition 22.}$$

(The name of this measure in Hitz and Montazeri (1995) is "C", not "Co". We use "Co" instead of "C" in order to avoid the name conflict with our set C of all classes in the system in Definition 1).

We always have $Co(c) \in [0, 1]$. Values 0 and 1 are taken for $|E_c| = |V_c| - 1$ and $|E_c| = |V_c|(|V_c| - 1)/2$, respectively.

4.1.4. Approach by Bieman and Kang (1995)

The approach by Bieman and Kang to measure cohesion is also based on that of Chidamber and Kemerer. They also consider pairs of methods which use common attributes. However, the manner in which an attribute may be used is different. A method m uses an attribute a directly, if $a \in AR(m)$. Method m uses attribute a indirectly, if m directly or indirectly invokes a method m' which uses attribute a: $\exists m' \in SIM^*(m)$: $a \in AR(m')$. Two methods are called "connected", if they directly or indirectly use a common attribute.

We define a predicate $cau(m_1, m_2)$ (common attribute usage) which is true, if $m_1, m_2 \in M_1(c)$ directly or indirectly use an attribute of class c in common:

$$cau(m_1, m_2) \Leftrightarrow \left(\bigcup_{m \in \{m_1\} \cup SIM^*(m_1)} AR(m) \right) \cap \left(\bigcup_{m \in \{m_2\} \cup SIM^*(m_2)} AR(m) \right) \cap A_I(c) \neq \emptyset$$

The measure TCC (tight class cohesion) is then defined as the percentage of pairs of public methods of the class with common attribute usage:

Definition 24 (measure TCC).

$$TCC(c) = 2 \frac{|\{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \cap M_{pub}(c) \wedge m_1 \neq m_2 \wedge cau(m_1, m_2)\}|}{|M_I(c) \cap M_{pub}(c) \mid (|M_I(c) \cap M_{pub}(c)| - 1)}$$

For the definition of LCC (loose class cohesion), let cau^* be the transitive closure of cau.

Definition 25 (measure LCC).

$$LCC(c) = 2\frac{|\{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \cap M_{pub}(c) \wedge m_1 \neq m_2 \wedge cau^*(m_1, m_2)\}|}{|M_I(c) \cap M_{pub}(c) \mid (|M_I(c) \cap M_{pub}(c)| - 1)}$$

With respect to inheritance, Bieman and Kang state three options for the analysis of the cohesion of a class:

(a) exclude inherited methods and inherited attributes from the analysis, or

(b) include inherited methods and inherited attributes in the analysis, or

(c) exclude inherited methods but include inherited attributes.

The above definitions of TCC and LCC conform to case (a). To define these measures according to case (b), we would have to use the sets $M(c)$ and $A(c)$ instead of $M_I(c)$ and $A_I(c)$ in the definitions of *cau*, *TCC* and *LCC*. For case (c), we only have to replace $A_I(c)$ by $A(c)$ in the definition of *cau*.

Bieman and Kang identified a problem with constructor methods for TCC and LCC. Constructor methods provide the class attributes with initial values and therefore access most or all of the class' attributes. If $mc \in M_I(c)$ is a constructor method which references all attributes of the class $(AR(mc) = A_I(c))$, then $cau(mc, m)$ is fulfilled for any method $m \in M_I(c)$ which references at least one attribute of class c $(AR(m) \neq \emptyset)$. That is, the presence of mc creates many pairs of directly connected methods. Furthermore, if m_1 and m_2 are two methods which reference at least one, but not necessarily the same, attribute of class c, then $cau(m_1, mc)$ and $cau(mc, m_2)$ are fulfilled, and thus $cau^*(m_1, m_2)$. That is, mc indirectly connects any two methods which use at least one attribute. We see that the presence of a constructor method artificially increases cohesion as measured by TCC and LCC, which is not empirically justified. Bieman and Kang therefore exclude constructors (and also destructors) from the analysis of cohesion (Bieman and Kang, 1995).

4.1.5. Approach by Henderson-Sellers (1996)

Henderson-Sellers sets out to define a cohesion measure having the following properties:

- The measure yields 0, if each method of the class references every attribute of the class (this situation is called "perfect cohesion" by Henderson-Sellers).
- The measure yields 1, if each method of the class references only a single attribute.
- Values between 0 and 1 are to be interpreted as percentages of the perfect value.

Henderson-Sellers proposes the following measure, which satisfies the above properties:

> Consider a set of methods $\{M_i\}$ $(i = 1, \ldots, m)$ accessing a set of attributes $\{A_j\}$ $(j = 1, \ldots, a)$. Let [...] the number of methods which access each datum be $\mu(A_j)$. [...]

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

Again, it is unclear whether inherited methods and attributes are accounted for or not. Using our formalism, we define this measure to be:

Definition 26 (measure LCOM5).

$$LCOM5(c) = \frac{|M_I(c)| - \dfrac{1}{|A_I(c)|}\displaystyle\sum_{a \in A_I(c)} |\{m \mid m \in M_I(c) \wedge a \in AR(m)\}|}{|M_I(c)| - 1}$$

4.1.6. Approach by Lee et al. (1995)

Lee et al. propose a set of cohesion measures based on information flow through method invocations within a class. For a method m implemented in class c, the cohesion of m is the number of invocations to other methods implemented in class c, weighted by the number of parameters of the invoked methods. The more parameters an invoked method has, the more information is passed, the stronger the link between the invoking and invoked method. The cohesion of a class is the sum of the cohesion of its methods. The cohesion of a set of classes simply is the sum of the cohesion of the classes in the set. Formally, these measures can be defined as follows:

Definition 27 (measure ICH).

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

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

$$ICH(SS) = \sum_{c \in SS} ICH(c).$$

4.1.7. Approach by Briand et al. (1993, 1994)

Briand et al. define a set of cohesion measures for object-based systems (such as Ada implementations). In the following, we adapt these measures to object-oriented systems. We make one simplification: the original measures were defined for so-called “software parts”, i.e., a module or a hierarchy of nested modules. We define the adapted object-oriented measures at the class level, but do not consider nested classes. Although some programming languages allow the definition of nested classes, nesting of classes is not a major issue in object-oriented design. It can be avoided by aggregation (defining attributes as an instance of another class), which is one of the most important object-oriented design concepts.

For the adaption of the cohesion measures to object-oriented systems, we see a class as a collection of *data declarations* and methods. Data declarations are (i) local, public type declarations (subset of $T(c)$), (ii) the class itself (as an implicit, public type), and (iii) public attributes (subset of $A_I(c)$, which also includes constants). A data declaration a *DD-interacts* with another data declaration b, if a change in a's declaration or use may cause the need for a change in b's declaration or use. We say there is a *DD-interaction* between a and b.

Examples:

- If the definition of a public type $t \in T(c)$ uses another public type $t' \in T(c)$, there is a DD-interaction between t' and t.
- If the definition of a public attribute $a \in A_I(c)$ uses a public type $t \in T(c)$, there is a DD-interaction between t and a.
- If a public attribute $a \in A_I(c)$ is an array and its definition uses public constant $a' \in A_I(c)$, there is a DD-interaction between a' and a.

DD-interactions need not be confined to one class. There can be DD-interactions between attributes and types of different classes. The DD-interaction relationship is transitive. If a DD-interacts with b and b DD-interacts with c, then a DD-interacts with c.

Data declarations also can interact with methods. There is a *DM-interaction* between data declaration a and method m, if a DD-interacts with at least one data declaration of m. Data declarations of methods include their parameters, return type and local variables. For instance, if a method m of class c takes a parameter of type class c, there is a DM-interaction between m and the implicit type declaration of class c.

All DD-interactions between data declarations, and DM-interactions involving parameters and return types can be determined from the class interface, and thus are available early in the development process. We define $CI(c)$ (CI for cohesive interactions) to be the set of all such DD- and DM-interactions. $Max(c)$ is the set of all possible DD- and DM-interactions in the class interface.

Definition 28 (measure RCI).

$$\text{For all classes } c \in C \text{ we define } RCI(c) = \frac{|CI(c)|}{|Max(c)|}.$$

RCI ranges between 0 and 1, where values 0 and 1 indicate minimum and maximum cohesion, respectively.

At the end of the high level design phase, designers will usually have a rough idea of which interactions there exist besides those that can be determined from the class interface. Three cases are possible:

(a) Some interactions will be known to exist. We will denote the set of all known interactions by $K(c)$. Notice that $CI(c) \subseteq K(c)$.

(b) Some interactions may or may not exist, the available information is not sufficient at the current development stage. We denote the set of these unknown interactions by $U(c)$.

(c) The remaining interactions are known not to exist.

Using this additional information, we can define three more measures:

Definition 29 (measures NRCI, PRCI, ORCI). For all classes $c \in C$ we define:

- The neutral ratio of cohesive interactions: $NRCI(c) = |K(c)|/(|Max(c)| - |U(c)|)$, (unknown interactions are not taken into account).
- The pessimistic ratio of cohesive interactions: $PRCI(c) = |K(c)|/|Max(c)|$, (unknown interactions are considered as if they were known not to be actual interactions).
- The optimistic ratio of cohesive interactions: $ORCI(c) = (|K(c)| + |U(c)|)/|Max(c)|$, (unknown interactions are considered as if they were known to be actual interactions).

4.2. Comparison of Approaches

A precise comparison of the approaches shows there are differences in the manner in which cohesion is addressed. One reason for this is the different objectives of the approaches. For example, Briand et al. examined only early design information to investigate potential early quality indicators while other authors investigated information mainly available at low level design and implementation; hence differences are found in the mechanisms that make a class cohesive. A second reason is that some of the issues dealt with by some authors are considered to be subjective and too difficult to measure automatically. For example, the degrees of method or class cohesion (addressed by Eder et al.) is not something which can be easily determined automatically or even manually. The following subsections discuss in detail the significant differences between the various approaches and what can be learned from these differences.

4.2.1. Types of Connection

By "type of connection" we refer to the mechanisms that link elements within a class and thus make a class cohesive. In the review of cohesion measures, we can distinguish two categories:

- In the first category, we find measures focused on counting pairs of methods that use or do not use attributes in common. Chidamber and Kemerer's idea of "similar" methods falls into this category; Hitz and Montazeri have reused this idea in their approach. The approach by Bieman and Kang also is based on counting pairs of methods that access attributes in common.

- In the second category, measures capture the extent to which individual methods use attributes or locally defined types (LCOM5, RCI), or invoke other methods (ICH).

It is possible to have one measure count different types of connections. For instance, measures LCOM4, TCC and LCC are focused on counting pairs of methods using common attributes, and method invocations.

The ICH suite of measures are based on method invocations solely. The attributes of a class are not considered at all. This is in sharp contrast to the definitions of all other measures.

4.2.2. Domain of the Measures

Most of the reviewed measures are defined at the class level. However, finer and coarser domains are also conceivable.

- For an individual attribute or method, we could count the number of other class elements to which it is connected, thus analyzing how closely related the attribute or method is to other elements of its class. This could also be interpreted as the degree to which the attribute or method contributes to the cohesion of its class. From such an analysis, we could draw conclusions as to how well the attribute or method "fits" into the class, or whether it should perhaps be moved to another class.
- We can quantify the cohesion of a set of classes or the whole system. This will be discussed in Section 5.2.

The ICH suite of measures is an example how a measure defined at the method level is scaled up to the class level and sets of classes. However, this done in a manner such that the measures are additive, which may not be a desirable property of a cohesion measure (see the theoretical validation of measures in Section 4.3.3 for further details).

4.2.3. Direct and Indirect Connections

Some of the approaches to measure cohesion include the analysis of indirectly connected elements. Indirect connections are of potential interest when defining criteria for when to break up a class. To illustrate this, we apply measures LCOM1 and LCOM3 to the example classes depicted in Figure 1. In the figure, a class c is represented by a graph G_c as in Definition 21 of measure LCOM3: the vertices are the methods of c, and there are edges between similar methods, i.e., methods which use an attribute in common. This is the type of connection both LCOM1 and LCOM3 are focused on. LCOM1 counts the number of pairs of methods in a class with no common attribute references. Because each class in Figure 1 has six methods and five pairs of similar methods, we have $\text{LCOM1}(c) = \text{LCOM1}(d)$, i.e., the classes are equally cohesive according to measure LCOM1. $\text{LCOM3}(c)$ is defined as the number of connected components of graph G_c. In Figure 1, it is $\text{LCOM3}(c) = 1$ and $\text{LCOM3}(d) = 2$, i.e., class c is more cohesive than class d according to measure LCOM3. This reflects an important difference between classes c and d: in class c, each method is

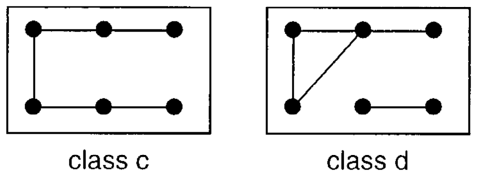


Figure 1. Example classes.

directly or indirectly connected with every other method. In class d, on the other hand, there are pairs of methods which are not even indirectly connected. This may indicate that the methods should not be encapsulated in the same class. Note, however, that there could be other reasons why the methods should be encapsulated together in one class anyway, e.g., because of method invocations from one connected component to the other.

Ideally, the graph G_c consists of only one connected component (LCOM3$(c) = 1$). Hitz and Montazeri remark, that class c can still be more or less cohesive. The number of edges of graph G_c can range between $n - 1$ (minimum cohesion) and $n(n - 1)/2$ (maximum cohesion), where $n = |M_I(c)|$ is the number of vertices of G_c. In other words, the discriminative power of measures counting the number of connected components (such as LCOM3 and LCOM4) is limited, because a connected component can show various degrees of connectivity. Therefore, Hitz and Montazeri proposed measure Co, which is a normalized count of the number of edges of G_c. Measure Co can be used to further discriminate classes for which graph G_c has only one connected component. However, using two measures to completely determine the cohesion of a class has the drawback that cohesion is no longer defined on an interval scale, but only on an ordinal scale. In addition, measure Co is not necessarily a better cohesion measure since it may not be possible to define classes with fully connected components.

TCC and LCC are also measures which take indirect connections into account, LCC even in two different ways. First, both measures count pairs of "connected" methods, i.e., methods which directly or indirectly use a common attribute. Method m uses an attribute a indirectly, if a is used by a method which is directly or indirectly invoked by m. Therefore, TCC and LCC take indirect method invocations into account. TCC counts the number of pairs of connected methods. It is therefore similar to measure Co, which counts the number of pairs of "similar" methods. LCC counts the number of directly or *indirectly* connected pairs of methods, and this is the second way in which indirect connections are accounted for by LCC. This again is related to the idea of counting connected components in LCOM3 or LCOM4: Consider a graph G where vertices are methods and there are edges between connected methods. Then, "two methods m and n are indirectly connected" is equivalent to "methods m and n lie within the same connected component of graph G". The condition "each method is directly or indirectly connected to every other method" is equivalent to "graph G consists of only one connected component". A low value of LCC corresponds with a large number of connected components of G. In that respect, LCC is conceptually similar to LCOM3 and LCOM4.

Hitz and Montazeri observe (Hitz and Montazeri, 1995) that a graph G_c consisting of more than one connected component indicates separable class cohesion according to the framework by Eder et al. (Section 4.1.1).

LCOM5 counts for each attribute how many methods access the attribute. Only direct connections between methods and attributes are considered. In a completely cohesive class, each attribute is accessed by every method. Whether such a design is desirable is unknown.

The RCI measures are a count of interactions between elements in the class. In a completely cohesive class, each element interacts with every other element. Because the interaction relationship is transitive, there need not be a direct interaction between all pairs of elements in order to have a maximum RCI. As a consequence, RCI does not have the drawback of LCOM5 that direct interactions between all elements are required to get a maximum value.

We summarize the results of this discussion:

- Indirect connections appear to be a better criterion than direct connections when indicators for when to split up a class are needed.
- With direct connections, each element of a class needs to be directly connected to every other element in order for the class to have maximum cohesion. This appears to be an unrealistic requirement.
- Measures accounting for indirect connections are less discriminative; maximum cohesion is possibly attained for a larger number of classes.

4.2.4. Inheritance

For the analysis of the cohesion of a class c, we have several options available concerning the attributes and methods c has inherited. Two straightforward options are:

(a) exclude inherited attributes and methods from the analysis, or

(b) include inherited attributes and methods in the analysis.

These two options form the distinction between class and inheritance cohesion in the framework by Eder et al. (see Section 4.1.1). A child class c represents an extension of its parent class d. If we exclude inherited attributes and methods, we analyze to what degree this extension represents a single semantic concept. If we include inherited attributes and methods, we analyze whether class c as a whole still represents a single semantic concept. These are two quite different aspects, and both should be considered.

Bieman and Kang offer a third option for the analysis of cohesion (Bieman and Kang, 1995):

(c) include inherited attributes, but exclude inherited methods from the analysis.

Bieman and Kang do not provide any rationale for this option.

A fourth alternative would be to exclude inherited attributes but include inherited methods. This of course makes little sense, as inherited methods can only access inherited attributes.

All of the measure definitions in Section 4.1 conform to case (a): inherited attributes and methods are excluded. However, with the exception of measures TCC, LCC and ICH, this is a result of our own interpretation of the measures' original definition. In the original definition of the RCI measures, inheritance is not addressed, because these measures were defined in context of object-based systems. In the original definition of all other measures, the influence of inheritance apparently has not been addressed.

Given the definitions in Section 4.1, we can easily derive new measures conforming to case (b): in the definitions, the sets $M_I(c)$ and $A_I(c)$ (non-inherited attributes and methods) simply have to be replaced by the sets $M(c)$ and $A(c)$ (all methods and attributes of the class).

4.2.5. Access Methods and Constructors

In object-oriented design, classes usually have "access methods". An access method provides read or write access to an attribute of the class. Access methods typically reference only one attribute, namely the one they provide access to. Thus, many pairs of access methods can be built, which do not use any common attributes. This constitutes a problem for measures which count such pairs (i.e., LCOM1, LCOM2, and LCOM3).

In addition, if other methods of the class use the access methods, they may no longer need to directly reference any attributes at all. Therefore, the presence of access methods artificially decreases the class cohesion for measures based on method-attribute references. In the definitions of LCOM4 and Co, this problem has been solved by adding method invocations to graph G_c, cf. Section 4.1.3. In the definitions of TCC and LCC, this problem is circumvented by introducing "indirectly" used attributes: if a method m invokes an access method, m indirectly uses the attributes accessed by that access method.

Constructor methods provide the class attributes with initial values and therefore access most or all of the class attributes. The presence of such a method constitutes a problem for measures counting "similar" or "connected" methods and indirect connections (LCOM3, LCOM4 and LCC). As explained in Section 4.1.4, the constructor method creates an indirect connection between any two methods which use at least one attribute, and artificially increases cohesion.

Destructors are less problematic, because they do not provide attributes with values and therefore do not need to reference all attributes.

4.2.6. Summary and Conclusions

From the above discussion we can see that there exists a variety of decisions to be made during the definition of a cohesion measure. It is important that decisions are based on the intended application of the measure if the measure is to be useful. When no decision for a particular aspect can be made, all alternatives should be investigated empirically. A second observation is that because the different aspects of cohesion are widely independent of each other, a large number of cohesion measures could be defined—this defines the problem space for cohesion measurement research in object-oriented systems.

4.3. Comparison of Measures

The comparison of cohesion measures and approaches is organized as follows. Section 4.3.1 introduces the criteria for comparing the measures. A summary of existing measures according to these criteria is provided in Section 4.3.2 and conclusions are then drawn about the state-of-the-art. Finally, Section 4.3.3 presents a theoretical validation of the cohesion measures and summarizes the results of this section.

4.3.1. Criteria of Comparison of Measures

In this section, we provide a list of criteria required to allow an initial comparison of measures to be performed and define the different levels of each criterion. These criteria then form the basic structure for the summary presented in Table 1.

- **Name**: The name of the measure.
- **Definition**: The definition of the measure using the defined formalism. The original definition of the measures are often ambiguous; hence, additional interpretation is required to define them using the formalism. We provide where necessary the most likely unambiguous alternative.
- **Operationally defined** (yes or no): Indicates if the original definition of the measure is operational or not, i.e., was additional interpretation of the measure's original definition necessary to come up with the definition of the measure given in column "Definition".
- **Objectivity** (subjective or objective): For an objective measure, the collected measurement data do not depend on the person collecting the data, i.e., the measure is automatable. For a subjective measure, the measurement data depend on the person collecting it and hence is not automatable.
- **Level of measurement** (nominal, ordinal, interval or ratio): The type of scale the measure is defined on. The type of scale is determined by the admissible transformations for the used empirical relation system (Fenton, 1991). However, rarely is the empirical relation system used for cohesion provided with the measure. In such cases, the indicated scale type reflects our intuitive judgment.
- **Partially usable** (An/HLD/LLD/Imp): This column and column "Usable" address the question when, in the development process, the measures become applicable. For this purpose, a generic object-oriented development process consisting of four development phases is used: analysis, high-level design, low-level design, and implementation. Details about these development phases can be found in Appendix C.
  A measure is classed as *partially usable* at the end of a development phase if the information required for the data collection is available at that phase, but is subject to refinement in later development phases. The column states the earliest development phase at which the measure is *partially usable*. Measurement values obtained at a

Table 1. Cohesion measures—overview.

Name	Definition	Operational definition	Objectivity	Level of measurement	Partially usable	Usable	Language-specific	Validation	Source
LCOM1 (lack of cohesion in methods)	$LCOM1(c) = \lvert\{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) = \varnothing\}\rvert$	no	obj	ratio	HLD	Imp	no	no	Henderson-Sellers (1996)
LCOM2	$LCOM2(c) = \begin{cases} \lvert P\rvert - \lvert Q\rvert, & if\ \lvert P\rvert > \lvert Q\rvert \\ 0, & otherwise \end{cases}$, where $P = \begin{cases} \varnothing, & if\ AR(m) = \varnothing\ \forall m \in M_I(c) \\ \{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) = \varnothing\}, & else \end{cases}$ $Q = \{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) \neq \varnothing\}$	no	obj	ord	HLD	Imp	no	th & emp	Chidamber and Kemerer (1994)
LCOM3	*LCOM3*(c) = number of connected components of $G_c=(V_c,E_c)$, where $V_c=M_I(c)$ and $E_c = \{\{m_1, m_2\} \mid m_1, m_2 \in V_c \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) \neq \varnothing\}$	no	obj	int	HLD	Imp	no	no	Chidamber and Kemerer (1991), Hitz and Montazeri (1995)
LCOM4	Like *LCOM3*(c), using modified set of edges $E_c = \{\{m_1, m_2\} \mid m_1, m_2 \in V_c \wedge (AR(m_1) \cap AR(m_2) \cap A_I(c) \neq \varnothing \vee m_1 \in SIM(m_2) \vee m_2 \in SIM(m_1))\}$	no	obj	int	HLD	Imp	no	no	Hitz and Montazeri (1995)
Co (connectivity)	$Co(c) = 2 \cdot \frac{\lvert E_c\rvert - (\lvert V_c\rvert - 1)}{(\lvert V_c\rvert - 1) \cdot (\lvert V_c\rvert - 2)}$, E_c and V_c are defined as in LCOM4	no	obj	ratio	HLD	Imp	no	no	Hitz and Montazeri (1995)
LCOM5	$LCOM5(c) = \frac{\lvert M_I(c)\rvert - \frac{1}{\lvert A_I(c)\rvert} \sum_{a \in A_I(c)} \lvert\{m \mid m \in M_I(c) \wedge a \in AR(m)\}\rvert}{\lvert M_I(c)\rvert - 1}$	no	obj	int	HLD	Imp	no	no	Henderson-Sellers (1996)

Table 1. Continued.

Name	Definition	Operational definition	Objectivity	Level of measurement	Partially usable	Usable	Language-specific	Validation	Source
TCC (tight class cohesion)	$TCC(c) = \frac{\lvert\{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \cap M_{pub}(c) \wedge m_1 \neq m_2 \wedge cau(m_1, m_2)\}\rvert}{\lvert M_I(c) \cap M_{pub}(c)\rvert(\lvert M_I(c) \cap M_{pub}(c)\rvert - 1)/2}$ where $cau(m_1, m_2) \Leftrightarrow \left(\bigcup_{m \in \{m_1\} \cup SIM^*(m_1)} AR(m)\right) \cap \left(\bigcup_{m \in \{m_2\} \cup SIM^*(m_2)} AR(m)\right) \cap A_I(c) \neq \varnothing$	yes	obj	ratio	HLD	Imp	no	no	Bieman and Kang (1995)
LCC (loose class cohesion)	$LCC(c) = \frac{\lvert\{\{m_1, m_2\} \mid m_1, m_2 \in M_I(c) \cap M_{pub}(c) \wedge m_1 \neq m_2 \wedge cau^*(m_1, m_2)\}\rvert}{\lvert M_I(c) \cap M_{pub}(c)\rvert(\lvert M_I(c) \cap M_{pub}(c)\rvert - 1)/2}$ where cau^* is the transitive closure of cau	yes	obj	ratio	HLD	Imp	no	no	Bieman and Kang (1995)
ICH (information-flow-based cohesion	$ICH^c(m) = \sum_{m' \in M_{NEW}(c) \cup M_{OVR}(c)} (1 + \lvert Par(m')\rvert) \cdot NPI(m, m')$ $ICH(c) = \sum_{m \in M_I(c)} ICH^c(m)$ $ICH(SS) = \sum_{c \in SS} ICH(c)$	yes	obj	ord	LLD	Imp	no	th	Lee et al. (1995)
RCI (ratio of cohesive interactions)	$RCI(c) = \frac{\lvert CI(c)\rvert}{\lvert Max(c)\rvert}$	yes	obj	ratio	HLD	Imp	no	th & emp	adapted from Briand, Morasca, and Basili (1994)
NRCI (Neutral RCI)	$NRCI(c) = \lvert K(c)\rvert / (\lvert Max(c)\rvert - \lvert U(c)\rvert)$	yes	obj	ratio	HLD	Imp	no	th	
PRCI (Pessimistic RCI)	$PRCI(c) = \lvert K(c)\rvert / \lvert Max(c)\rvert$	yes	obj	ratio	HLD	Imp	no	th	
ORCI (Optimistic RCI)	$ORCI(c) = (\lvert K(c)\rvert + \lvert U(c)\rvert) / \lvert Max(c)\rvert$	yes	obj	ratio	HLD	Imp	no	th	

development phase where the measure is *partially usable* are only approximations; their values are likely to change in subsequent development phases.

- **Usable** (An/HLD/LLD/Imp): A measure is *usable* at a given development phase if all information required for data collection is available and stable, i.e., the information is refined only to a limited extent in subsequent development phases. We state the earliest development phase at which the measure is *usable*.
- **Language specific**: If the measure is specific to a particular programming language, the language is provided. If a measure is language specific, this does not imply that the measure is not applicable to other languages, but adapting the measure will be necessary before it can be applied to other languages.
- **Validation** (th, emp, no): Indicates if and how the measure has been validated. There is a distinction between:
  - Theoretical validation (th): The authors have validated their measure theoretically, usually by analyzing its mathematical properties. The analysis and results can be found in the first publication referenced in the "source" column (see next item on this list).
  - Empirical validation (emp): The measure has been used in an empirical validation investigating its causal relationship on an external quality attribute. For these measures, the validation results are discussed in Section 6.0.
- **Source**: Literature references where the measure has been proposed.

4.3.2. Overview of Measures

An overview of the cohesion measures introduced in Section 4.1 is presented in Table 1. Before discussing the individual measures in detail a number of simple, but important observations can be made.

First, there is no measure in Table 1 which is classified "usable" at the analysis or HLD phases. For measures classified "partially usable" at analysis or HLD only approximations of the values that are obtained at LLD or implementation can be computed. Empirical studies are required to analyze how accurate such approximations are and whether they are useful predictors of external attributes.

Second, the original definitions of many of the measures are not fully operational, i.e., additional interpretation has been required to formalize these measures. Above all, the measures are imprecise with respect to inheritance.

Third, many measures are neither validated theoretically or, more importantly, empirically. Consequently, little evidence exists to support the notion that the cohesion measures are actually useful in terms of predicting relevant external product quality attributes. To strengthen software measurement research and to convince practitioners of the usefulness of software measures, a larger number of such validation studies must be performed.

4.3.3. Theoretical Validation

In this section, we theoretically validate the cohesion measures in Table 1 with respect to four cohesion properties defined by Briand, Morasca, and Basili (1996). The motivation behind defining such properties is that a measure must be supported by some underlying theory—if it is not, then the usefulness of that measure is questionable. The four cohesion properties defined by Briand et al. are one of the more recent proposals to characterize cohesion in a reasonably intuitive and rigorous manner. While these properties are not sufficient to say that a measure which fulfills them all will be useful, it is likely that a measure which does not fulfill them all is ill-defined.

In the following discussion, let *Cohesion* be a candidate measure for the cohesion of a class or an object-oriented system. Relationships capture the connections within a class on which the respective cohesion measure is focused. Consider, for instance, a measure counting pairs of similar methods (methods which use an attribute in common). Then there is a relationship between any two methods which use an attribute in common. For a class c, the set of relationships within the class is denoted by R_c. The set of all intra-class relationships in an object-oriented system is defined as $IR = \bigcup_{c \in C} R_c$. We say R_c is maximal, if all possible relationships within class c are present, i.e., it is not possible to add a relationship to class c and thus obtain a new class c' such that $R_c \subset R_{c'}$, $R_c \neq R_{c'}$. We say IR is maximal, if R_c is maximal $\forall c \in C$.

Cohesion.1. Nonnegativity and Normalization. The cohesion [of a class c | of an object-oriented system C] belongs to a specified interval:

$$[Cohesion(c) \in [0, Max] \mid Cohesion(C) \in [0, Max]].$$

Cohesion.2. Null value and maximum value. The cohesion [of a class c | of an object-oriented system C] is null if [R_c | IR] is empty, and the cohesion [of a class c | of an object-oriented system C] is *Max* if [R_c | IR] is maximal:

$$[R_c = \emptyset \Rightarrow Cohesion(c) = 0 \mid IR = \emptyset \Rightarrow Cohesion(C) = 0]$$
$$[R_c = maximal \Rightarrow Cohesion(c) = Max \mid IR\, maximal \Rightarrow Cohesion(C) = Max].$$

Cohesion.3. Monotonicity. Let C be an object-oriented system, and $c \in C$ be a class in C. We modify class c to form a new class c' which is identical to c except that $R_c \subseteq R_{c'}$, i.e., we added some relationships in c. Let C' be the object-oriented system which is identical to C except that class c is replaced by class c'. Then

$$[Cohesion(c) \leq Cohesion(c') \mid Cohesion(C) \leq Cohesion(C')].$$

Cohesion.4. Merging of unconnected classes. Let C be an object-oriented system, and $c_1, c_2 \in C$ two classes in c. Let c' be the class which is the union of c_1 and c_2. Let C' be the object-oriented system which is identical to C except that classes c_1 and c_2 are replaced by c'. If no relationships exist between classes c_1 and c_2 in C, then

$$[max\{Cohesion(c_1), Cohesion(c_2)\} \geq Cohesion(c') \,|\, Cohesion(C) \geq Cohesion(C')].$$

Cohesion.3 says that if a relationship is added to the system, cohesion must not decrease. Cohesion.4 says that merging two unconnected classes must not increase cohesion (because the union of these classes has little cohesion).

The LCOM measures LCOM1 to LCOM5 are inverse cohesion measures: a low value indicates high cohesion and vice versa. Properties Cohesion.2 to Cohesion.4 are not appropriate for these measures. In order to apply these properties to the LCOM measures, we have to adapt the properties' definitions. Let *Lack_of_Cohesion* be a candidate measure for the (inverse) cohesion of a class or an object-oriented system.

Cohesion.2′. *Lack_of_Cohesion* [of a class c | of an object-oriented system C] is zero if $[R_c \mid IR]$ is maximal, and it is *Max*, if $[R_c \mid IR]$ is empty:

$$[R_c = \emptyset \Rightarrow Lack\_of\_Cohesion(c) = Max \mid IR = \emptyset \Rightarrow Lack\_of\_Cohesion(C) = Max]$$
$$[R_c\ maximal \Rightarrow Lack\_of\_Cohesion(c) = 0 \mid IR\ maximal \Rightarrow Lack\_of\_Cohesion(C) = 0].$$

Cohesion.3′. Adding a relationship to the system must not increase *Lack_of_Cohesion*:

$$[Lack\_of\_Cohesion(c) \geq Lack\_of\_Cohesion(c') \mid Lack\_of\_Cohesion(C) \geq Lack\_of\_Cohesion(C')].$$

Cohesion.4′. Merging two unconnected modules must not decrease *Lack_of_Cohesion*:

$$[min\{Lack\_of\_Cohesion(c_1), Lack\_of\_Cohesion(c_2)\} \leq Lack\_of\_Cohesion(c') \mid Lack\_of\_Cohesion(C) \leq Lack\_of\_Cohesion(C')].$$

Results from Theoretical Validation

There are only a few cohesion measures which fulfill all of the cohesion properties. These are: TCC, LCC, and the RCI measures. All other measures violate one or more cohesion properties.

- The following measures are not normalized (Cohesion.1): LCOM1, LCOM3, LCOM4, LCOM2 and ICH. There is no upper limit of the values that these measures can take. Normalization is intended to allow for comparison of the cohesion of classes and systems of different size. Without normalization, this is not possible.
- LCOM5 has been normalized to range between 0 and 1 under the assumption that each attribute of a class is referenced by at least one method. Of course, having an attribute

which is not used by any methods is silly. However, in the course of maintenance activities, such situations may occur, and a cohesion measure should be prepared for it. If we drop the assumption that each attribute is referenced by at least one method, LCOM5 yields values between 0 and 2. It is still normalized, however, the maximum value 2 is taken only if $M_I(c) = 2$. For $|M_I(c)| > 2$, the maximum possible value is smaller. This can be seen as follows: LCOM5 takes its maximum value if there are no references to attributes at all. In this case, the value of LCOM5 is $m/(m-1)$, where $m = |M_I(c)|$ is the number of methods of c. Thus, the maximum value of LCOM5 is 2 for $m = 2$, $3/2$ for $m = 3$, $4/3$ for $m = 4$, etc. This is an anomaly. If we redefine this measure as

$$NewCoh(c) = \begin{cases} 0, \text{ if } M_I(c) = \emptyset \text{ or } A_I(c) = \emptyset \\ \frac{\sum_{a \in A_I(c)} |\{m | m \in M_I(c) \wedge a \in AR(m)\}|}{|M_I(c)||A_I(c)|}, \text{ otherwise} \end{cases}$$

we get a cohesion measure which is conceptually similar to LCOM5, but normalized properly (and not an inverse cohesion measure: large values indicate high cohesion, small values indicate low cohesion).

- The connectivity measure Co has been normalized to range between 0 and 1 under the assumption that graph G_C consists of exactly one connected component. If we drop this assumption, Co can yield negative values. Therefore, Co violates properties Cohesion.1 and Cohesion.2. If we redefine Co as

$$Co'(c) = 2 \cdot \frac{|E_C|}{|V_C| \cdot (|V_C| - 1)},$$

all four cohesion properties are satisfied.

- LCOM2 is not monotonic (Cohesion.3′): In the pathological case that the methods of a class c do not reference any of the attributes of c, we have $LCOM4(c) = 0$ (because $P = \emptyset$ by definition, and $Q = \emptyset$). If we change c such that exactly one method of c references exactly one attribute of c, we get $LCOM4(c) > 0$, since P is no longer empty. Because LCOM2 is an inverse cohesion measure, this means that the modified class is less cohesive than the original class. If we drop the case where P is set to $\emptyset$ in the definition of P, this anomaly disappears.

- ICH violates Cohesion.4. If we merge two unconnected classes c_1 and c_2 to form a new class c, we get $ICH(c) = ICH(c_1) + ICH(c_2)$. That is, the cohesion of c is rated higher than that of c_1 and c_2.

In the remainder of this section, we point out problems with individual measures which are not covered by the above cohesion properties.

Measure LCOM2 is known to have little discriminating power. This is partly due to the fact that LCOM2(c) is set to zero whenever there are more pairs of methods which use an attribute in common than pairs of methods which do not. As a result, LCOM2 is zero for

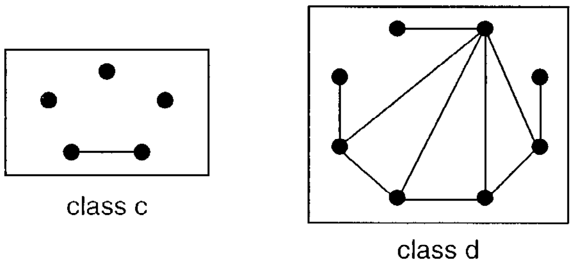


Figure 2. Two different classes with equal values for LCOM2.

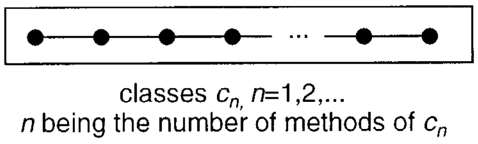

classes c_n, n=1,2,...
n being the number of methods of c_n

Figure 3. Similar classes with different values for LCOM2.

a large number of classes (Basili, Briand, and Melo, 1996). But also for classes where LCOM2 is greater than zero, the measure is not discriminating. Consider the example classes c and d in Figure 2, proposed by Henderson-Sellers (1996) (the symbols in the figure have the same semantics as in Figure 1). For class c, we have $|P| = 9$, $|Q| = 1$, and thus LCOM2$(c) = 8$. For class d, it is $|P| = 18$, $|Q| = 10$, and so LCOM2$(d) = 8$. Both classes have the same LCOM2 value, but we would intuitively say that class d is more cohesive than class c.

Similarly, Hitz and Montazeri (1995) provide an example of a set of classes c_n (see Figure 3), which are structurally very similar but yield different values LCOM2(c_n). The values depend on parameter n, the number of methods in the class. For example, for $n < 5$ we have LCOM2$(c_n) = 0$, for $n = 5, 6, 7$, and 8 LCOM2(c_n) takes values 2, 5, 9, and 14, respectively.

Table 2 summarizes the results of this section. Column "Cohesion Criteria" indicates for each measure what makes a class cohesive according to the measure. "Indirect connections" indicates if the measure accounts for transitive dependencies in the class. "Inheritance" specifies how inheritance has been treated in the original definition of the measure (n.a. stands for "not addressed"). "Known problems" summarizes any problems we have identified for the measure. Columns C.1 to C.4 indicate violations of the above cohesion properties Cohesion.1 to Cohesion.4 (for inverse cohesion measures, columns C.2 to C.4 indicate violations of properties Cohesion.2′ to Cohesion.4′). Violations of properties are marked with an "X".

Table 2. Cohesion measures—specific properties.

Measure	Cohesion Criteria	Indirect connections	Inheritance	Known Problems	C.1	C.2	C.3	C.4
LCOM1	sharing of attributes	no	n.a.	access methods	X			
LCOM2	sharing of attributes	no	n.a.	access methods, not discriminating: no negative values	X		X	
LCOM3	sharing of attributes	yes	n.a.	access methods, constructors	X			
LCOM4	sharing of attributes, method invocations	yes	n.a.	constructors	X			
Co	sharing of attributes, method invocations	no	n.a.		X	X		
LCOM5	attribute usage	no	n.a.	access methods, optimal cohesion is bad design	X			
TCC	sharing of attributes, method invocations	no	three alternatives offered					
LCC	sharing of attributes, method invocations	yes	three alternatives offered	constructors				
ICH	method invocations	no	exclude inherited methods		X			X
RCI	type and attribute usage	yes	n.a. (originally defined for object-based systems)					
NRCI	type and attribute usage	yes	n.a. (originally defined for object-based systems)					
PRCI	type and attribute usage	yes	n.a. (originally defined for object-based systems)					
ORCI	type and attribute usage	yes	n.a. (originally defined for object-based systems)					

5.0. A Unified Framework for Cohesion Measurement

In this section, a new framework for cohesion in object-oriented systems is proposed. The framework is defined on the basis of the issues identified by comparing the various approaches to measure cohesion (Section 4.2) and the discussion of existing measures. The objective of the unified framework is to support the comparison and selection of existing cohesion measures with respect to a particular measurement goal. In addition, the framework should provide guidelines to support the definition of new measures with respect to a particular measurement goal when there are no existing measures available. The framework, if used as intended, will

- ensure that measure definitions are based on explicit decisions and well understood properties,
- ensure that all relevant alternatives have been considered for each decision made,

- highlight dimensions of cohesion for which there are few or no measures defined.

The framework consists of five criteria, each criterion determining one basic aspect of the resulting measure. First, we describe each criterion: what decisions have to be made, what are the available options, how is the criterion reflected by the cohesion measures in Section 4.1. We then discuss how the framework can be used to derive cohesion measures. For each criterion, we have to choose one or more of the available options which will be strongly influenced by the stated measurement goal.

The five criteria of the framework are:

1. The type of connection, i.e., what makes a class cohesive.
2. Domain of the measure.
3. Direct or indirect connections.
4. Inheritance: how to assign attributes and methods to classes, how to account for polymorphism.
5. How to account for access methods and constructors.

These criteria are necessary to consider when specifying a cohesion measure. However, they are not sufficient, as other aspects such as properties of measures (e.g., those proposed in Briand, Morasca, and Basili (1996)) and results from empirical validation studies have to be considered too. The influence of these aspects is not addressed here.

We now describe each of the criteria in the order given above.

5.1. Framework Criteria

5.1.1. Type of Connection

By type of connection we mean the mechanism that makes a class cohesive. In Table 3 we summarize types of connections used by the measures in Section 4.1. A connection within a class is a link between elements of the class (attributes, methods, or data declarations). For each type of connection, the elements are listed in the columns “Element 1” and “Element 2”. Column “Description” explains the type of connection. Column “Design phase” indicates from which design phase on the type of connection is typically applicable. Column “Measures” lists for each type of connection, which of the reviewed measures use that type of connection. The numbers in column “#” are used later to reference the types of connections.

5.1.2. Domain of the Measure

The domain of the measure specifies the objects to be measured (methods, classes etc.). Table 4 shows possible domains for the cohesion measures, and for each domain, the measures from Section 4.1 having that domain.

Table 3. Types of connection.

#	Element 1	Element 2	Description	Design phase	Measures
1	method m of class c	attribute a of class c	m references a	HLD	LCOM5
2	method m of class c	method m' of class c	m invokes m'	HLD	ICH, LCOM4, Co
3	method m of class c	method m' of class c, $m \neq m'$	m and m' directly reference an attribute a of class c in common ("similar methods")	HLD	LCOM1, LCOM3, LCOM4, Co, LCOM2
4	method m of class c	method m' of class c, $m \neq m'$	m and m' directly or indirectly reference an attribute a of class c in common ("connected methods")	HLD	TCC, LCC
5	data declaration in class c	data declaration in class c	data-data interaction	HLD	RCI, NRCI, ORCI, PRCI
6	method m of class c	data declaration in class c	data-method interaction	HLD	RCI, NRCI, ORCI, PRCI

As we see, most measures are defined at the class level. Measures defined at the attribute and method level are also conceivable. These measures count the number of connections a method or attribute has to other elements of the class. Measures defined on the class level can be scaled up to sets of classes or the whole system.

5.1.3. Direct or Indirect Connections

We have to decide whether to count direct connections only or also indirect connections. For example, consider a method m_1 which is "similar" to a method m_2 (connection type #3), which in turn is similar to method m_3. Then methods m_1 and m_2 are directly connected

Table 4. Mapping of measures to domains.

Domain	Measures
attribute	—
method	ICH
class	LCOM1, LCOM2, LCOM3, Co, LCOM4, LCOM5, TCC, LCC, ICH, RCI, NRCI, PRCI, ORCI
set of classes	ICH
system	—

Table 5. Measures counting direct and indirect connections.

Type	Measures
direct	LCOM1, Co, LCOM2, LCOM5, TCC, ICH
indirect	LCOM3, LCOM4, LCC, RCI, NRCI, ORCI, PRCI

through a connection of type #3, as are methods m_2 and m_3. Methods m_1 and m_3 are indirectly connected.

Table 5 shows which measures in Section 4.1 count direct connections only and which also count indirect connections.

5.1.4. Inheritance

Two aspects are to be considered with respect to inheritance:

- How do we assign methods and attributes to classes?
- For method invocation: shall we consider static or polymorphic invocations?

The aspects can be dealt with in the order they are listed here.

a) How to assign methods and attributes to classes

As we found in the review of the cohesion measures, we have two options available concerning the attributes and methods a class c has inherited for the analysis of the cohesion of c:

(a) Exclude inherited attributes and methods from the analysis.

A child class c represents an extension of its parent class d. If we exclude inherited attributes and methods, we analyze to what degree this extension represents a single semantic concept.

(b) Include inherited attributes and methods in the analysis.

If we include inherited attributes and methods, we analyze whether class c as a whole still represents a single semantic concept.

All of the measures as they are defined in Section 4.1 conform to option (a). Again, note that for most measures this is due to our own interpretation, as this aspect has not been dealt with in the original measures' definitions.

b) Polymorphism

The next question is how to deal with polymorphism. This is relevant only if the chosen type of connection involves method invocations (types #2 and #4), for the special case that a

Table 6. Mapping of measures to options for accounting for polymorphism.

Type	Measure
account for polymorphism	ICH
do not account for polymorphism	LCOM4, Co, TCC, LCC

method of a class c contains a polymorphistic method invocation of a method implemented at an ancestor of class c. We have two options:

- Account for polymorphism, i.e., for a method m, we consider invocations of all methods $m' \in PIM(m)$.
- Do not account for polymorphism, i.e., for a method m, we consider invocations of methods $m' \in SIM(m)$ only.

Table 6 shows which measures in Section 4.1 account for polymorphism and which do not. Only measures counting connection of types #2 and #4 are considered in the table.

5.1.5. How to Account for Access Methods and Constructors

As we have seen in the review of the measures, access methods and constructors may artificially increase or decrease the values for cohesion measures. How to account for access methods and constructors should be a conscious decision in the definition of a cohesion measure and is therefore part of the framework.

a) Access Methods

Access methods cause problems for measures which count references to attributes (connection types #1 and #3). Instead of referencing an attribute directly, the access method may be used, which is not accounted for by these types of connections. Thus, the number of references to attributes is artificially decreased. A solution to this problem is to count the invocation of an access method as reference to the attribute. However, this solution may be difficult to implement in practice because it is not always possible to recognize access methods automatically.

Access methods also cause problems for measures that count pairs of methods which use common attributes (connection types #3 and #4). Because access methods usually access only one attribute, many pairs of methods that do not reference a common attribute can be formed using access methods. Thus, the cohesion is artificially decreased. A solution to this problem is to exclude access methods from the analysis.

The available options for how to deal with access methods are summarized in Table 7. Column "Connections" indicates the types of connections for which the respective option is applicable.

The measures as defined in Section 4.1 all conform to option 1.

Table 7. Option to account for access methods.

Option	Description	Connections
1	Do nothing (treat access methods as regular methods)	All types
2	Consider the invocation of an access method as a reference to that attribute	#1, #3
3	Exclude access methods from the analysis	#3, #4

Table 8. Options to account for constructors.

Option	Description	Connection
4	Do nothing (treat constructors as regular methods).	All types
5	Exclude constructors from the analysis	#3, #4

b) Constructors

Constructors cause problems for measures that count pairs of methods which use attributes in common (connection types #3 and #4). Constructors typically reference all attributes. This artificially increases the cohesion of the class, because it generates many pairs of methods that use an attribute in common. A solution to this problem is to exclude constructors from the analysis. We thus have two options how to account for constructors, which are summarized in Table 8.

The measures in Section 4.1 all conform to option 4; for measures TCC and LCC, application of option 5 has been suggested.

5.2. Application of the Framework

We apply the framework to select existing measures or to derive new measures for a given measurement goal. Note that the framework is not intended to be used as a means to search cohesion measures in an ad hoc manner, or to generate an exhaustive set of theoretically possible cohesion measures. Application is performed by following two steps:

- For each criterion of the framework, choose one or more of the available options basing each decision on the objective of measurement.
- Choose the existing measures that match the decisions made, or, if none exist, construct new cohesion measures. Remember that properties such as those presented in Section 4.3.3 can also be used to guide the definition and theoretical validation of new measures.

In the context of applying this framework, the measurement goal must at least specify the underlying hypothesis which drives measurement. The hypothesis will be of the form “Internal attribute cohesion (as measured by the cohesion measures to be defined) has a causal effect on external quality attribute Y.” The external attribute Y could be maintainability, reliability etc. As discussed in Briand, El Emam, and Morasca (1995), we believe that product measures by themselves, no matter how well defined, are not guaranteed to capture any relevant phenomenon regarding the quality of the system under study. It must be shown empirically that they are related to some external quality attribute of interest. In other words, it is crucial to provide evidence that they are relevant quality indicators in order to be used and relied upon.

It is recommended to first define measures for the external attribute in the hypothesis and then apply the framework to derive cohesion measures. Having an operational definition of the external quality attribute may help in the processes of choosing the appropriate cohesion measures.

Construction and Normalization of the Cohesion Measures

As a result of the application of the criteria 1, 3, 4, and 5, we have identified the set of connections in a class that are of interest to us. In criterion 2 we determined the domain of the measure. We now describe how to construct the normalized cohesion measures accordingly.

For the definition of the measure, we need two figures: the number of actual connections of interest, and the maximum number of possible connections of interest. For measures defined for attributes and methods, these figures include the connections of interest in which the attribute or method actually participates respectively can participate. For measures defined for classes, the figures include the connections of interest in the class (actual and possible connections). For sets of classes and the whole system, the figures include connections of interest in all relevant classes (actual and possible).

If we define the cohesion measure as

$$\frac{\textit{number of actual connections of interest}}{\textit{maximum number of possible connections of interest}}$$

we get a measure which is normalized to range between 0 and 1.

Since we normalize the measures, we cannot count multiple connections between elements separately. For instance, a method can reference the same attribute several times, these are multiple connections between the method and the attribute. If we counted multiple connections separately, the maximum number of possible connections would be infinite. The ICH measures in Section 4.1 count individual connections, i.e., multiple invocations of the same method are counted separately. Therefore, the ICH measures are not normalized and cannot be normalized.

5.3. Summary

We conclude the discussion of the unified framework with the following remarks.

- The measures generated with this framework are proportions of the maximum possible number of connections within classes. This leads to the to the highest level of measurement, the ratio level, which means the most powerful types of statistical analysis techniques can be performed.

- These measures, however, are not guaranteed to be useful. To be useful, the measures must be empirically validated with respect to the external quality attribute of interest specified in the measurement goal. We believe that measures of internal product attributes have no inherent significance in isolation. They become useful only if they are related to external quality attributes.

- Existing measures have been classified according to the options available for each criterion of the framework. This classification allows existing measures to be compared. The classification has shown that some particular options of the framework criteria have no or only few corresponding measures proposed.

6.0. Empirical Validation Studies

In this section, we discuss empirical studies that have been performed with the reviewed cohesion measures. In Section 6.1, we focus on studies conducted to empirically validate measures, i.e., show the usefulness of the measures. These are studies where at least one of the measures discussed in Section 4.1 has been used to investigate the relationship of internal attribute cohesion to an external quality attribute of a software product.

We briefly present the results of these studies and analyze their validity. In Section 6.2, we give a brief overview of empirical work other than validation that has been performed using the reviewed cohesion measures. Section 6.3 summarizes the results of this section.

6.1. Empirical Validation Studies

We are aware of only three publications that empirically validate some of the measures presented in Section 4.1. In Section 6.1.1, we describe the systems and dependent variables used in the studies in more detail. In Section 6.1.2 we presents the results from the analyses, and in Section 6.1.3 we analyse the validity of these results.

6.1.1. Systems and Dependent Variables Used

The systems and dependent variables used in the studies are described in Table 9.

6.1.2. Results of the Studies

Li and Henry (1993)

In Li and Henry (1993), measures of the suite by Chidamber and Kemerer (1991) were tested, including measure LCOM3. Li and Henry proposed several multivariate regression models and conducted least-square regression and F-tests to estimate their predictive power.

Table 9. Description of systems and dependent variables.

Publication	Li and Henry (1993)	Basili, Briand, and Melo (1996)	Briand, Morasca, and Basili (1994)
Systems	two commercial systems (UIMS, QUES)	eight systems from a student's project, developed by eight teams in four months	three industrial systems (GOADA, GOESIM, TONS)
Application domain of systems	UIMS: User Interface System QUES: Quality Evaluation System	Information system for video rental businesses	GOADA: ground support, simulator, navigation for satellites
Design method	Classic-Ada design language (object-oriented extension of Ada)	OMT (Rumbaugh et al. 1991)	- no information available -
Implementation language	Classic-Ada programming language	C++	Ada
Size	39 classes (UIMS), 70 classes (QUES)	total of 180 classes in all eight systems, sizes of systems between 5 and 14 KSLOC	GOADA: 90 KSLOC / 525 Ada units GOESIM: 170 KSLOC / 676 Ada units TONS: 50 KSLOC / 180 Ada units
Developers	- no information available -	eight groups of three students, had already some experience with C or C++, no previous experience with object-oriented analysis/design	- no information available -
Dependent Variable	Maintenance effort: number of lines changed in a class over a period of 3 years, calculated as follows: each deleted line counts 1, each added line counts 1, each modified line counts 2 (one deletion and one addition)	Fault-proneness: were faults detected in a class during acceptance testing (yes or no). Each system underwent eight hours of acceptance testing, detected faults were then traced back to classes.	Fault-proneness: were faults detected in a unit during acceptance testing (yes or no).
Independent Variable	Chidamber and Kemerer's metrics suite (Chidamber and Kemerer, 1991)	Chidamber and Kemerer's metrics suite (Chidamber and Kemerer, 1994)	RCI
Modeling technique	linear least-square regression	logistic regression	logistic regression

Unfortunately, Li and Henry only reported the R^2, adjusted R^2, and p-values from the F-tests for their regression models, but—with one exception—not the regression coefficients and p-values for each independent variable. We therefore cannot draw any conclusions whether LCOM3 has been found to be a significant predictor for maintenance effort. For one multivariate model, the regression coefficients are given. For system UIMS, the regression coefficient for LCOM3 is 2.762436; for system QUES, it is -2.195476. That is, increasing

LCOM3 suggests an increase in maintenance effort in UIMS (this is the trend expected by the authors), but a decrease in maintenance effort in QUES. The unexpected trend in QUES is not explained by Li and Henry.

Basili et al. (1996)

In Basili et al. (1996), the measures of the suite by Chidamber and Kemerer (1994) were tested, including LCOM2. Univariate logistic regression was performed to test the predictive power of each measure in isolation. Then, several multivariate models were built in a stepwise selection process, and tested.

LCOM2 is the only measure that has not been found significant both in the univariate and multivariate analysis. The measurement values of LCOM2 showed small variance, so that this measure could not be used as predictor. Basili et al. attributed this to the definition of LCOM2: the measurement value is set to zero whenever there are more pairs of methods in a class which use common attributes than pairs of methods which do not. As a result, LCOM2 is zero for a large number of classes which otherwise would yield (different) negative values for LCOM2.

Briand et al. (1994)

Briand et al. tested measure RCI (more precisely, a version for object-based systems which also considers nested modules) and some measures for coupling. For each system GOADA, GOESIM, and TONS, univariate logistic regression was performed to test the predictive power of each measure in isolation. Then, a multivariate model was built for each system in a stepwise selection process, and tested. Table 10 summarizes the results for measure RCI from these analyses. Column "C" gives the regression coefficient, column "p" the p-value, i.e., the statistical significance, determined by a likelihood ratio test. The value in column "$\Delta\psi$" for univariate analysis is an evaluation of the impact of the measure on the dependent variable. $\Delta\psi$ is based on the notion of odds ratio. The odds ratio $\psi(X)$ represents the ratio between the probability of having a fault and not having a fault when the value of the measure is X. $\Delta\psi$ is defined as $\Delta\psi = \psi(X+1)/\psi(X)$, i.e., $\Delta\psi$ represents the reduction or increase in the odds ratio (expressed in the table as a percentage) when the value X increases by one unit.

6.1.3. Threats to Validity

We distinguish between three types of threats to validity of an empirical study:

- Construct validity: The degree to which the independent and dependent variables accurately measure the concepts they purport to measure.
- Internal validity: The degree to which conclusions can be drawn about the causal effect of the independent variables on the dependent variables.

Table 10. Results for measure RCI.

	GOADA			GOESIM			TONS		
	C	$\Delta\psi$	p	C	$\Delta\psi$	p	C	$\Delta\psi$	p
Univariate Analysis	0.63	19%	0.000	0.215	12%	0.047	0.34	14%	0.001
Multivariate Analysis	0.4		0.006	0.3		0.07	0.2		0.16

- External validity: The degree to which the results of the research can be generalized to the population under study and other research settings.

We now apply these criteria to the studies described above.

Construct Validity

The choice of the dependent variable used in Li and Henry (1993) to measure maintainability (or maintenance effort) is questionable: The "number of lines changed in a class" may say little about the actual effort (in terms of person-hours etc.) spent on the maintenance of a class: some changes to a class may require more effort than others, even though they contribute the same or even less to the count of "number of lines changed", and vice versa.

Internal Validity

In Briand, Morasca, and Basili (1994) and Basili, Briand, and Melo (1996), the function of some measures as "design measures" is emphasized, i.e., these measures are applicable in early stages of the development process. If these measures are found to be useful, they can be used to detect problems in the design before implementation starts, thus potentially saving time and effort for rework of the design. However, in the studies measurement was performed only after implementation. If measurement had taken place, say, before implementation started, different measurement data would have been obtained, because the description of the system before implementation is less complete than after implementation. This in turn could have led to different results in the statistical analyses. The validity of the measures in early development phases is therefore still to be confirmed. In order to demonstrate the usefulness of measures in early development phases, the measures must be applied to the deliverables of the early design phases.

External Validity

Basili et al. list the following facts that may restrict the generalizability of their results (Basili, Briand, and Melo, 1996).

- The systems developed are rather small: they lie between 5000 and 14000 source lines of C++ code.
- The systems developed have a limited conceptual complexity.
- The developers may not be as well trained and experienced as average professional programmers.

The first point (rather small systems) also applies to the studies in Li and Henry (1993): system UIMS has 39 classes, system QUES has 70 classes. We have no information concerning the conceptual complexity of the systems and the experience of the developers for the studies in Li and Henry (1993) and Briand, Morasca, and Basili (1994).

6.2. Other Empirical Studies

In this section, we briefly present other empirical work that has been performed with the measures in Section 4.1.

Bieman and Kang (1995) analyzed the impact of cohesion as measured by TCC and LCC on private reuse, i.e., reuse via inheritance and reuse via instantiation. Reuse via inheritance for a class *c* has been defined as the number of classes that inherit from *c*. Reuse via instantiation for a class *c* has been defined as the number of other classes where the class *c* is instantiated (for instance through aggregation, or if an object of class *c* is created in a method of another class). The authors found no relation between cohesion and reuse via instantiation. However, classes with a higher number of descendents tended to have lower cohesion. This is contrary to what the authors expected to find. They did not provide an explanation for this trend. The usefulness of this study is limited, since the dependent variable, private reuse, is a measure of an *internal* attribute. Internal attributes represent no externally visible qualities of the system. They do not have any inherent meaning or usefulness unless they are seen in relationship with some external attribute.

In Chidamber and Kemerer (1994), the measures proposed in that paper were applied to two systems. The distribution of the measurement values was presented and an ad hoc interpretation of the data was done (e.g., explanations for differences in the distribution of the measurement values between the two systems). This study showed that the measures are collectable in large systems. However, we cannot draw any conclusion about the usefulness of these measures from this discussion.

6.3. Summary

In Basili, Briand, and Melo (1996), LCOM2 was not found to be a significant predictor of fault-prone classes. From the description of the studies in Li and Henry (1993), we cannot tell whether LCOM3 was a significant indicator for the number of lines changed in a class. The only successful validation reported is for measure RCI (Briand, Morasca, and Basili, 1994) as a predictor of fault-proneness in an object-based programming environment. This small amount of empirical validation work that has been published is of concern. More empirical work is definitely required.

One reason for the small amount of empirical validation work may be that measures are often provided without underlying empirical models; without these, there can be no hypothesis testing. Another reason is that for empirical validation we need to define and measure an internal and an external attribute, and measuring an external quality attribute can in practice be difficult. Measuring an internal attribute is relatively easy: measures are quickly defined and, in order to apply the measure, we only need access to the artifacts of some already existing systems. Typically, we have some systems available and can perform the measurement. Measuring an external quality attribute is more problematic, because we need additional information besides the measured system. For instance, if we want to measure maintenance effort, we need a system for which we have the appropriate maintenance effort data. Typically, we do not have systems for which the additional required information is available, since few organizations have adequate measurement programs in place.

Trade-offs in the definition of the external attribute to circumvent these problems limit the usefulness of the study. For instance, in Li and Henry (1993) maintenance effort has been defined as the number of lines changed in a class. One advantage is that the number of lines changed in a class can be calculated if we have several versions of the same system, i.e., we do not need actual effort data. However, as already highlighted, the number of lines changed is not a straightforward indicator of the actual effort spent on the maintenance of the class.

7.0. Conclusions

Based on a standardized terminology and formalism, we have provided a framework for the comparison, evaluation, and definition of cohesion measures in object-oriented systems. This framework is intended to be exhaustive and integrates new ideas with existing measurement frameworks in the literature. Thus, detailed guidance is provided so that cohesion measures may be defined in a consistent and operational way and existing measures may be selected based on explicit criteria.

We have also used this framework to review the state-of-the-art, about which we draw the following conclusions:

- There is a very rich body of ideas regarding the way to address cohesion measurement in object-oriented systems.
- However, many measures are not based on explicit empirical models and, therefore, their intended application is a priori difficult to determine.
- Very few measures have undertaken a thorough empirical validation. In other words, the usefulness of many of the measures is currently not empirically supported.
- When empirical validations do exist, they are sometimes seriously flawed because of threats to the validity of their results, e.g., construct validity when the dependent variable used in the analysis does not capture accurately the external quality attribute of interest.

Therefore, it appears that, although many good ideas have been reported, there is too little empirical work in (cohesion) measurement, especially in the context of object-oriented systems. This can only hinder effective research and the design of satisfactory solutions for the practitioners of measurement.

Future work includes the identification of guidelines describing how—in the course of applying the framework to define or select cohesion measures—the measurement goal at hand influences the choice of the available options for each criterion of the framework. To this end, we also plan to perform an exhaustive empirical investigation of the measures reported in this article.

Acknowledgments

The research reported here has been conducted within the framework of Jürgen Wüst's master's thesis on quality measures for object-oriented systems.

Appendix

A. Glossary of Terms

Standard terminology used for object-oriented concepts is provided as follows:

- Component: Any system entity whose properties may be measured. Most important are typically classes, methods, and attributes.
- Class: Compound structure encapsulating data and functional elements.
- Object: Instance of a class.
- Attribute: A data item encapsulated in a class. Other names: instance variable, data member, state variable.
- Method: A procedure or function encapsulated in a class. Other names: operation, service, member function.
- Inheritance: "Is-a" relationship between two classes. A class c may inherit from class d. The methods and attributes of class d are then available to class c.
- Parent class: If class c inherits from class d, class d is a parent class. Other name: superclass.
- Child class: If class c inherits from class d, class c is a child class. Other name: subclass.
- Descendent class: The descendent classes of a class c are the children classes of class c, their children etc. (any class that directly or indirectly inherits from class c).
- Ancestor class: The ancestor classes of a class c are the parent classes of class c, their parent classes etc. (any class, from which c directly or indirectly inherits).

- Signature: Unique identifier that identifies a method. The signature specifies the method's name, the parameters it takes, and the return type.
- Interface: The set of all signatures defined as public within a class, i.e., the interface characterizes the complete set of messages that can be sent to an instance of that class.
- Body: The body of a method is its implementation. The body of a class is the implementation of its methods.
- Access Method: A method whose sole purpose is to provides access to one or more attributes of the class.
- Constructor: A method which creates and initializes an object of a class.
- Virtual method: A method which has no implementation. The implementation of the method is deferred to children classes.
- Abstract class: A class which has at least one virtual method. No objects can be instantiated from an abstract class.
- Message: Classes, via their methods, send messages to request services from other classes, possibly including specific recipients and parameters.
- Polymorphism: An identifier may refer to instances of different classes (typically having a common ancestor) at run-time, allowing objects to be bound to this identifier to respond to the same set of messages in different ways (however, the semantics of the response should be similar for all objects).

Also defined is the applicable measurement terminology. The definition of the terms "measure", "internal attribute", "external attribute", "theoretical validation" and "empirical validation" are taken from Briand, El Emam, and Morasca (1995):

- Measure, domain, range: Let D be a set of empirical objects to be measured (e.g., a set of classes, methods), and R be a set of formal objects (e.g., real numbers). A measure is a mapping μ: $D \rightarrow R$, which maps every element of D onto an element of R. We call D the *domain* of measure μ, R the *range* of μ. Other name: metric.
- Internal attribute: A quality or property of a software product that can be measured in terms of the product itself, e.g., size, coupling, etc.
- External attribute: A quality or property of a software product that can not be measured solely in terms of the product itself. For instance, to measure maintainability of a product, measurement of maintenance activities on the product will be required in addition to measurement of the product itself.
- Operationally defined: A measure is considered operationally defined if no further interpretation of its definition is required to use it, i.e., it is stated in an unambiguous manner.

```
class c {
  public:
    virtual void m1()=0;
    void m2(int);
    void m3(char);
};

void c::m2(int i) {/*...*/}
void c::m3(char ch) {/*...*/}
```

```
class d : public c {
  public:
    void m1();
    void m3(char);
    void m4();
};

void d::m1() {/*...*/}
void d::m3(char ch) {/*...*/}
void d::m4() {/*...*/}
```

Figure 4. Example methods.

- Theoretical validation: A demonstration that a measure is really measuring the internal or external attribute it purports to measure.
- Empirical validation: A demonstration that a measure is useful in the sense that it is related to an interesting external attribute in an expected way.
- Measurement goal: Specification of the objectives of measurement in a given context. In this paper, it is assumed that the objective of measurement is to test a hypothesis of the form: "Internal attribute X has an impact on external attribute Y", i.e., to conduct an empirical validation. Other information that typically is included in the measurement goal: the development phase at which measurement is to take place, the environment in which measurement is to take place (company, development team, methodology used etc.), properties for measures of internal or external attributes.

B. Explaining the Formalism

This appendix provides illustrating examples for some of the definitions used in the formalism provided in Section 3.0.

Declared and Implemented Methods

In Definition 3, set $M_D(c)$ of methods *declared in c* and set $M_I(c)$ of methods *implemented in c* were defined.

Consider the example classes in Figure 4. For class c, $M(c) = \{c::m1, c::m2, c::m3\}$ where $M_D(c) = \{c::m1\}$ and $M_I(c) = \{c::m2, c::m3\}$. For class d, because of inheritance, $M(d) = \{d::m1, c::m2, d::m3, d::m4\}$ with $M_D(d) = \{c::m2\}$ and $M_I(d) = \{d::m1, d::m3, d::m4\}$.

Notice that unique labels, e.g., "$c::m1$", are used to represent any given method. In the case where a class inherits a method and does not override it, the method retains the identity provided by the parent class. In the example, class d inherits method $c::m2$ of class c and does not override it. Thus, $M_I(c) \cap M_D(d) \neq \emptyset$ because $c::m2 \in M_I(c)$

and $c::m2 \in M_D(d)$. In the other cases each method is provided with a new identity. For example, $d::m3$, $d::m4 \notin M(c)$. In addition, certain object-oriented programming languages allow "method overloading" where a class can have two or more methods with the same name but different signatures. Such methods will still be provided with a unique identity.

Inherited, Overriding and New Methods

In Definition 4, sets $M_{INH}(c)$, $M_{OVR}(c)$ and $M_{NEW}(c)$ of inherited, overriding and new methods of a class c were introduced. In the example in Figure 4 it is $M_{NEW}(d) = \{d::m4\}$, $M_{OVR}(d) = \{d::m1, d::m3\}$ and $M_{INH}(d) = \{c::m2\}$.

For any class $c \in C$ and method $m \in M_{NEW}(c)$ there is no method with the same signature declared in any ancestor class of c. $M_{INH}(c)$, $M_{OVR}(c)$ and $M_{NEW}(c)$ form a partition of $M(c)$: they are pairwise disjoint, and $M_{INH}(c) \cup M_{OVR}(c) \cup M_{NEW}(c) = M(c)$.

Method Invocations

Consider the example in Figure 5. We make the following remarks:

- Method $c::mc1$ invokes method $d::md$ using a pointer to an object of type d. Class d is the "static type" of the object being pointed to. At run-time, the dynamic type of the object may be of class d or any descendent class of d. Due to polymorphism, the actual method being executed can be implemented in class d or any descendent class of d (in the example, either $d::md$ or $d1::md$ may be executed; $d2::md$ is identical to $d::md$).

- The body of $c::mc1$ contains a total of three statements which invoke method $d::md$. How often the method $d::md$ is actually invoked at run-time, however, cannot be determined from static analysis.

In the following, we show the values for *SIM*, *NSI*, *PIM* and *NPI* from Definitions 8 to 11 for method $c::m1$ in Figure 5. It is $SIM(c::mc1) = \{d::md\}$ and $NSI(c::mc1, d::md) = 3$. As a result of inheritance, the same method can be statically invoked for objects of different classes. Method $d::md$ is statically invoked twice for an object of type d, and once for an object of type $d2$. It is $PIM(c::mc1) = \{d::md, d1::md\}$, where $NPI(c::mc1, d::md) = 3$ and $NPI(c::mc1, d1::md) = 2$. As a result of polymorphism, one method invocation can contribute to the NPI count of several methods.

Notice that the static type of an object for which a method m' is invoked may be an abstract class and m' may be a virtual method. The dynamic type of an object cannot be an abstract class because an object cannot be an instance of an abstract class.

```
class c {
     d *d_ptr;
     d2 d_obj;
  public:
     void mc1(int);
}

void c::mc1(int j) {
  d_ptr->md(-1);
  for( int i=1; i<=j; i++)
     d_ptr->md(i);
  d_obj.md()
}
```

```
class d {
  public:
     virtual void md(int);
};
void d::md(int i) {/*...*/}

class d1 : public d {
  public:
     void md(int);
};
void d1::md(int i) {/*...*/}

class d2 : public d {
  public:
     void md2();
};
```

Figure 5. Example method invocation.

```
class c {
  public:
     int x,y,z;
     /* ... */

};
```

```
class d : public c {
     int a;
     const int b=2;
  public:
     int f();

  /* ... */
};

int d::f() {
  a=a+1; x=a; y=b; return a+b;
}
```

Figure 6. Example class with attributes.

Declared and Implemented Attributes

To illustrate declared and implemented attributes (Definition 13), consider the example in Figure 6. It is $A_D(c) = \emptyset$ and $A_I(c) = A(c) = \{c::x, c::y, c::z\}$. For class d $A_I(d) = \{d::a, d::b\}$, $A_D(d) = \{c::x, c::y, c::z\}$, and $A(d) = \{d::a, d::b, c::x, c::y, c::z\}$.

Like methods, attributes are modeled as elements of sets and require a unique identity. As with methods, non-inherited attributes of a class are provided with their own identity, whereas inherited attributes are provided with their identity in some ancestor class. Notice no distinction is made between constant attributes and regular attributes. For instance, for class d, $d::b \in A(d)$.

Attribute References

In Definition 15, set *AR* was introduced to model the set of attributes referenced by a method. In Figure 6, it is $AR(d::f) = \{d::a, c::x, c::y, d::b\}$. Notice there is no distinction between read and write accesses to attributes as none of the measures discussed make this distinction.

C. A Generic Object-Oriented Development Process

In the survey of measures in Section 4.2, we classify the measures according to when during the development process they become applicable. To be able to classify the measures consistently, it must be known when certain deliverables required for measurement are available. To achieve this requirement, a generic development process with four steps and the deliverables available at the end of each is defined. These deliverables comprise of modeling concepts which are similar to those used by most object-oriented methodologies and are exemplified by means of a mapping to Jacobson's OOSE method (Jacobson et al., 1992). In general, each step will be performed in several iterative cycles, the deliverables being updated as the problem and solution are more clearly defined.

- Analysis (An): The following deliverables are available at the end of the analysis phase:

  – High level classes: High level classes model the entities in the problem domain. A high-level class (HLC) will in later phases be implemented by one or more regular classes, i.e., classes as they are provided by programming languages. At the analysis phase we know nothing about the internal structure of HLCs. We do have a good idea of the services the HLC provides.

  – Inheritance relationships: we have knowledge of some inheritance relationships between HLCs, derived mainly as identification of "type-of" relationships. In general, the number of inheritance relationships identified during analysis will be relatively small.

  – Other relationships: These are relationships between HLCs such as "uses", "consists-of", etc. These relationships are derived on the basis of the services a HLC is to provide. For example, if HLC *A* requests a service which is provided by HLC *B*, there is a uses relationship between *A* and *B*.

  Note that it is also usual for the system to be decomposed into subsystems, i.e., groups of closely related HLCs. This occurs for ease of understandability and iterative enhancement.

  Mapping to OOSE: In the following, terms specific to the OOSE terminology are set in quotes to distinguish them from our standard terminology. The analysis phase corresponds to the "Robustness analysis" in the OOSE method. The artifacts described above are those found in the "Analysis model": HLCs are "objects" ("interface, entity or control objects"); inheritance relationships have their direct counterpart in OOSE; the other relationships are called “associations” (“communication associations, acquaintance associations”). The services provided by each HLC (i.e., “object” in OOSE) are not part of the “Analysis model”, but are evident from the “use cases” defined in an earlier process step of OOSE.

Most object-oriented methodologies feature an early analysis phase and introduce a graphical notation, where high-level classes are represented by boxes (or circles), and relationships (inheritance, uses, etc.) are represented by different kinds of arrows between circles or boxes). The information contained in these diagrams is considered to be the measurable output of the analysis.

– High-level design (HLD): During high-level design, the HLCs are refined. This involves the following decisions: which regular classes are needed to implement a high-level class, what methods must a HLC provide, what input and output parameters will the methods need, in which “regular” class should each method be implemented, what data will each class hold. Also, we will know which functionality each method has to fulfill, and have a rough idea about which other methods a method uses. The methods at this level are “high-level” methods. Several methods may be required later to implement one “high-level” method, that is, new methods will be added later. Also, the input and output parameters are still subject to later refinement.

  As the refinement of the HLCs creates new classes, new inheritance relationships between classes will arise. For instance, if we identified a number of classes which perform network communication, these classes are likely to have some functionality (methods) in common (e.g., wait for message, send message, receive message). This functionality could be factored out in a common parent class of the network communication classes.

  Mapping to OOSE: The high-level design corresponds to the first half of the “Construction” phase of the OOSE method. The HLCs are mapped onto “blocks”, and each block consists of one or more classes (the implementation environment will influence the choice of classes). Using “interaction diagrams”, “stimuli” between blocks are determined, and what information is passed with each “stimulus”. The “stimuli” correspond to the “high-level” methods, the information passed corresponds to the parameters.

– Low-level design (LLD): During low-level design, algorithms for each method are designed. Typically, techniques such as state-transition graphs, flowcharts, or program description languages (PDL) are used. The design of algorithms, as well as determining the precise signature for each method, is likely to identify the need for new methods and attributes. There is also detailed information about which methods and attributes are used by any given method.

  Further possibilities for class abstraction can still be discovered at LLD and the use of library classes is considered. This can result in some new classes being added to the system and minor rearrangement of the inheritance hierarchy.

Mapping to OOSE: The low-level design phase corresponds to the second half of the "Construction" phase in OOSE. State-transition graphs are used to design algorithms for the methods of each class.

– Implementation (Imp): After implementation the source code is available. Mapping to OOSE: OOSE has a process step "Implementation" which produces the source code.

D. Levels Of Cohesion As Defined By Myers

In this appendix, we restate the definition of cohesion given by Myers (1978) in the context of structured design. In the following, the term "module" refers to a set of one or more contiguous program statements having a name by which other parts of the system can invoke it. Myers identified six levels of cohesive modules. Listed from lowest (worst) to tightest (best) cohesion, these are:

- *Coincidental*: A module has coincidental cohesion if its function cannot be defined, or if it performs multiple, completely unrelated functions.
- *Logical*: The module performs a set of related functions, e.g., a module that performs all input and output operations for the program. Which function is performed on invocation is determined by an argument passed to the module.
- *Temporal*: The module performs multiple sequential functions which are weakly related (for instance modules that do all the "initialization" or "termination clean-up").
- *Procedural*: The module performs multiple sequential functions, where the sequential relationship between all of the functions is implied by the specification of the module.
- *Communicational*: The module performs multiple sequential functions, where the sequential relationship between all of the functions is implied by the specification of the module, and there is a data relationship among all of the functions (all functions operate on the same set of data).
- *Functional*: The module performs a single specific function (transformation of some input data to some output data).

Notes

1. Note that this figure includes variations of the same measure, e.g., there are four different versions of the LCOM (lack of cohesion in methods) measure originally proposed by Chidamber and Kemerer (1991).

References

Basili, V. R., Briand, L. C., and Melo, W. L. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering* 22(10): 751–761.

Bieman, J. M., and Kang, B.-K. 1995. Cohesion and reuse in an object-oriented system. *Proc. ACM Symp. Software Reusability (SSR'94)*, 259–262.

Briand, L., Daly, J., and Wüst, J. 1996. A unified framework for coupling measurement in object-oriented systems. Fraunhofer Institute for Experimental Software Engineering, Germany, Technical Report ISERN 96-14.

Briand, L., El Emam, K., and Morasca, S. 1995. Theoretical and empirical validation of software product measures. Technical Report ISERN 95-03.

Briand, L., Morasca, S., and Basili, V. 1993. Measuring and assessing maintainability at the end of high-level design. *IEEE Conference on Software Maintenance*, Montreal, Canada.

Briand, L., Morasca, S., and Basili, V. 1994. Defining and validating high-level design metrics. University of Maryland, CS-TR 3301, Technical Report.

Briand, L., Morasca, S., and Basili, V. 1996. Property-based software engineering measurement. *IEEE Transactions of Software Engineering* 22(1): 68–86.

Bunge, M. 1977. *Treatise on Basic Philosophy: Ontology I: The Furniture of the World*. Boston: Riedel.

Bunge, M. 1979. *Treatise on Basic Philosophy: Ontology II: The World of Systems*. Boston: Riedel.

Card, D. N., Church, V. E., and Agresti, W. W. 1986. An empirical study of software design practices. *IEEE Transactions on Software Engineering* 12(2): 264–271.

Card, D. N., Page, G. T., and McGarry, F. E. 1985. Criteria for software modularization. *Proceedings IEEE Eighth International Conference on Software Engineering*, 372–377.

Chidamber, S. R., and Kemerer, C. F. 1991. Towards a metrics suite for object oriented design. In *Proc. Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91)* (A. Paepcke, ed.), SIGPLAN Notices 26(11): 197–211.

Chidamber, S. R., and Kemerer, C. F. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20(6): 476–493.

Churcher, N. I., Shepperd, M. J. 1995a. Comments on 'A metrics suite for object-oriented design.' *IEEE Transactions on Software Engineering* 21(3): 263–265.

Churcher, N. I., and Shepperd, M. J. 1995b. Towards a conceptual framework for object oriented software metrics. *Software Engineering Notes* 20(2): 69–76.

Coad, P., and Yourdon, E. 1991a. *Object-Oriented Analysis*, second edition. Prentice Hall.

Coad, P., and Yourdon, E. 1991b. *Object-Oriented Design*, first edition. Prentice Hall.

Eder, J., Kappel, G., and Schrefl, M. 1994. Coupling and cohesion in object-oriented systems. University of Klagenfurt, Technical Report.

Embley, D. W., and Woodfield, S. N. 1987. Cohesion and coupling for abstract data types. *6th International Phoenix Conference on Computers and Communications*, Arizona.

Fenton, N. 1991. *Software Metrics: A Rigorous Approach*. Chapman and Hall.

Henderson-Sellers, B. 1996. *Software Metrics*. Hemel Hempstaed, U.K.: Prentice Hall.

Hitz, M., and Montazeri, B. 1995. Measuring coupling and cohesion in object-oriented systems. *Proc. Int. Symposium on Applied Corporate Computing*, Monterrey, Mexico.

Hitz, M., and Montazeri, B. 1996. Chidamber & Kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering* 22(4): 276–270.

Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: ACM Press/Addison-Wesley.

Li, W., and Henry, S. 1993. Object-oriented metrics that predict maintainability. *J. Systems and Software* 23(2): 111–122.

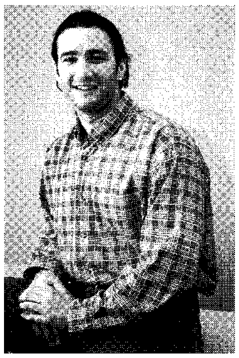
Lee, Y.-S., Liang, B.-S., Wu, S.-F., and Wang, F.-J. 1995. Measuring the coupling and cohesion of an object-oriented program based on information flow. *Proc. International Conference on Software Quality*, Maribor, Slovenia.

Myers, G. 1978. *Composite/Structured Design*. Van Nostrand Reinhold.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. *Object-Oriented Modeling and Design*. Prentice Hall.

Lionel C. Briand received the B.S. degree in geophysics and the M.S. degree in Computer Science from the University of Paris VI, France. He received the Ph.D. degree, with high honors, in Computer Science from the University of Paris XI, France.

Lionel is currently the head of the Quality and Process Engineering Department at the Fraunhofer Institute for Experimental Software Engineering (FhG IESE), an industry-oriented research center located in Rheinland-Pfalz, Germany. His current research interests and industrial activities include measurement and modeling of software development products and processes, software quality assurance, domain specific architectures, reuse, and reengineering. He has published numerous articles in international conferences and journals and has been a PC member or chair in several conferences such as ICSE, ICSM, ISSRE, METRICS, and SEKE. Before that, Lionel started his career as a software engineer at CISI Ingénierie, France. He then joined, as a research scientist, the NASA Software Engineering Laboratory, a research consortium: NASA Goddard Space Flight Center, University of Maryland, and Computer Science Corporation. Before going to FhG IESE, he held the position of lead researcher of the software engineering group at CRIM, the Computer Research Institute of Montreal, Canada.



John W. Daly received the B.Sc. and Ph.D. degrees in Computer Science from the University of Strathclyde, Glasgow, Scotland in 1992 and 1996 respectively. He is currently a software engineering researcher in the Fraunhofer Institute (FhG IESE), Kaiserslautern, Germany where he has been employed since April, 1996.

John's current research interests and industrial activities include software measurement and software process improvement, conducting empirical research by means of quantitative and qualitative methods, and object-oriented development techniques.

Jürgen Wüst received the degree Diplom-Informatiker (M.S.) in Computer Science with a minor in Mathematics from the University of Kaiserslautern, Germany, in 1997. He is currently a researcher at the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern, Germany. His current research activities and industrial activities include software measurement, software architecture evaluation, and object-oriented development techniques.