A Unified Framework for Coupling Measurement in Object-Oriented Systems

Lionel C. Briand, John W. Daly, and Jürgen K. Wüst

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 coupling measurement in object-oriented systems. However, despite a very interesting and rich 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 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 frameworks and measures for coupling measurement 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.

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 that many measures are not defined in a fully operational form, and relatively few of them are based on explicit empirical models, as recommended by measurement theory.

Index Terms—Coupling, object-oriented, measurement.

1 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 coupling. High quality software design, among many other principles, should obey the principle of low coupling. Stevens et al., who first introduced coupling in the context of structured development techniques, define coupling as "the measure of the strength of association established by a connection from one module to another" [35]. Therefore, the stronger the coupling between modules, i.e., the more inter-related they are, the more difficult these modules are to understand, change, and correct and thus the more complex the resulting software system. Some empirical evidence exists to support this theory for structured development techniques; see, e.g., [33], [36].

The principle of low coupling has now been migrated to object-oriented design by Coad and Yourdon [18], [19] and recent research has again led to a large number of new coupling measures for object-oriented systems being defined. However, because coupling is a more complex software attribute in object-oriented systems (e.g., there are many different mechanisms that can constitute coupling) 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 coupling 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 coupling measures means the usefulness of most measures is not supported.

To address and clarify our understanding of the state-of-the-art of coupling measurement in object-oriented systems requires a comprehensive framework based on a standard terminology and formalism. This framework can then be used:

1) to facilitate comparison of existing measures,
2) to facilitate the evaluation and empirical validation of existing measures, and
3) to support the definition of new measures and the selection of existing ones based on a particular goal of measurement.

- *L.C. Briand and J.K. Wüst are with the Fraunhofer Institute for Experimental Software Engineering (IESE), Sauerwiesen 6, D-67661 Kaiserslautern, Germany. E-mail: {briand, wuest}@iese.fhg.de.*
- *J.W. Daly is with Hewlett Packard Ltd., QA Department, R&D, Queensferry Microwave Division, South Queensferry, EH30 9TG Scotland. E-mail: dalyj@sqf.hp.com.*

Manuscript received 5 Mar. 1997; revised 9 Apr. 1998.
Recommended for acceptance by D. Perry.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 104095.

Analogous research for cohesion measurement is described in [8]. The cohesion framework presented in that paper is complementary to the coupling framework presented here.

The majority of the work on coupling measurement in object-oriented systems focuses on usage dependencies between classes, which can be derived from a static analysis of design documents or source code. The dynamic aspects of coupling between objects at run-time have barely been investigated and are not yet considered in practice. Even though we address this issue in the survey, we make no attempt to integrate it in our framework and restrict ourselves to the analysis of static usage dependencies in a system.

The paper is organized as follows. Section 2 summarizes the current state of coupling measurement in object-oriented system and provides detailed motivation for the need for the research performed in this paper. Section 3 introduces the notation and formalism required to conduct this research. Section 4 provides a comprehensive review and structured synthesis of existing object-oriented coupling frameworks and measures. The results of this review are then used to define a new unified framework for coupling measurement in object-oriented systems in Section 5.

2 Motivation

Object-oriented measurement has become an increasingly popular research area. This is substantiated by the fact that recently proposed in the literature are:

1) several different frameworks for coupling and cohesion, and
2) 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 coupling measure—these decisions have to be made with respect to the goal of the measure and by defining an empirical model based on clearly stated hypotheses. Unfortunately, for many measures proposed in the literature these decisions and hypotheses are not documented. It is, therefore, often unclear what the potential uses of existing measures are and how different coupling measures could be used in a complementary manner to obtain a more detailed picture of the coupling in an object-oriented system. In short, our understanding of existing coupling measures is not what it should be.

Several authors have tried to address this problem by introducing frameworks to characterize different approaches to coupling and the relative strengths of these, although, on their own, none of the frameworks could be considered comprehensive. There are three existing and quite different frameworks for object-oriented coupling (reviewed in detail in Section 4.1). First, Eder et al. identify three different types of relationships [20]. These relationships, interaction relationships between methods, component relationships between classes, and inheritance between classes, are then used to derive different dimensions of coupling which are classified according to different strengths. Second, Hitz and Montazeri approach coupling by deriving two different types of coupling: object level coupling and class level coupling which are determined by the state of an object and the state of an object's implementation, respectively [22]. Again different strengths of coupling are proposed. And third, Briand et al. constitute coupling as interactions between classes [4]. The strength of the coupling is determined by the type of the interaction, the relationship between the classes, and the interaction's locus of impact. As none of the frameworks have been used to characterize existing measures to the different dimensions of coupling identified, the negative aspects highlighted above are still very prevalent ones. In our review of the literature, for example, we found more than 30 different measures[1] of object-oriented coupling. 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 coupling measurement we have to integrate all existing frameworks into a unique theoretical framework, based on a homogenous and comprehensive formalism. A review has to be performed of existing measures 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 coupling measures by ensuring that specific hypotheses are provided which link coupling measures to external quality attributes. It should also facilitate identification of dimensions of coupling which thus far have been neglected, i.e., for which there are no measures defined. Finally, the framework must be able to integrate new coupling measures as they are defined in the future. In that sense both the formalism and the framework must be extensible.

3 Terminology and Formalism

In the past, research within the area of software measurement has suffered from a lack of:

1) standardized terminology, and
2) 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 [16] and Hitz and Montazeri [24] have identified ambigui-

1. Note that this figure includes variations of the same measure, e.g., there are three versions of the RFC (response for a class) measure originally proposed by Chidamber and Kemerer [14].

ties in members of the well referenced suite of measures by Chidamber and Kemerer [15]. 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 coupling 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 coupling in object-oriented systems and to measurement in general. This can be referenced as required. Where appropriate the terminology defined by Churcher and Shepperd [17] has been used.

To express the coupling measures consistently and unambiguously the following formalism based on set and graph 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. Otherwise, simple explanations and examples are provided in [9].

3.1 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*

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

3.2 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 nonvirtual and either inherited, overridden, or newly defined, all of which have implications for measuring coupling. 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 nonvirtual noninherited methods of c.*
- *where $M(c) = M_D(c) \cup M_I(c)$ and $M_D(c) \cap M_I(c) = \phi$.*

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 noninherited, nonoverriding methods of c.*

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

DEFINITION 5 ($M(C)$. The Set of all Methods). *$M(C)$ 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 coupling measurement, must be defined.

DEFINITION 6 (Parameters). *For each method $m \in M(C)$ let $Par(m)$ be the set of parameters of method m.*

3.3 Method Invocations

To measure coupling 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. For static invocations, the invoked method is determined by the type of the variable that references the object for which the method invocation occurred. For dynamic invocations, the invoked methods are determined by considering all possible types that the object for which the method invocation occurred may have at run-time. Consequently, for each method $m \in M(c)$ the following sets are defined.

DEFINITION 7 ($SIM(m)$. 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 8 ($NSI(m, m')$. 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 9 ($PIM(m)$. 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 and dynamic binding, be invoked for an object of dynamic type d.*

DEFINITION 10 ($NPI(m, m')$. 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

for a nonvirtual method $m' \in M(C)$, if $m' \in SIM(m)$ for some $m \in M(C)$ then it also is $m' \in PIM(m)$. For a virtual method $mv' \in M(C)$, we can have $mv' \in SIM(m)$, but we could not have $mv' \in PIM(m)$ (because mv' is virtual, its class is abstract, therefore no object of that class can be instantiated).

3.4 Attributes

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

DEFINITION 11 (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., noninherited attributes).*

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

DEFINITION 12 ($A(C)$. The Set of all Attributes). *$A(C)$ is the set of all attributes in the system and is represented as $A(C) = \bigcup_{c \in C} A(c)$.*

3.5 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 13 ($AR(m)$). *For each $m \in M(C)$ let $AR(m)$ be the set of attributes referenced by method m.*

3.6 Types

Attributes and parameters have types which all can contribute to coupling. 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).

DEFINITION 14 (Basic Types and 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 (e.g., records, enumerations, but not classes).*

- 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 15 ($\boldsymbol{T}$ The Set of Available Types). *The set $\boldsymbol{T}$ of available types in the system is $\boldsymbol{T} = BT \cup UDT \cup C$.*

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

DEFINITION 16 (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.

3.7 Predicates

To ensure consistency the predicate *uses* must be defined.

DEFINITION 17 (Uses). Let $c \in C$, $d \in C$.

$$uses(c, d) \Leftrightarrow (\exists m \in M_I(c): \exists m' \in M_I(d): m' \in PIM(m)) \vee (\exists m \in M_I(c): \exists a \in A_I(d): a \in AR(m))$$

A class *c* uses a class *d* if a method implemented in class *c* references a method or an attribute implemented in class *d*.

3.8 C++ Specific Extensions

In C++, a class *c* can declare a class *d* its friend, which means that *d* is granted access to nonpublic elements of *c*. We must be able to specify for a class *c*, which are the friends of class *c* and which classes declare class *c* their friend, as this is likely to have an influence on the strength of coupling between the classes.

DEFINITION 18 (Friend). *For each class $c \in C$, we define the set $Friends(c) \subset C$ of friend classes of c. For each class $c \in C$, the set of inverse friends of c (i.e., the set of classes that declare c their friend) is defined as:*

$$Friends^{-1}(c) = \{d \mid d \in C \wedge c \in Friends(d)\}.$$

In C++, a class can also declare an individual method its friend. However, none of the coupling measures scrutinize friendship at this level of detail. Therefore, we do not model friendship relationships between classes and methods.

In hybrid languages such as C++, which also provide features found in the procedural paradigm, it is possible to take a pointer of a method, and pass this pointer to another method. This also is a type of coupling considered by some measures, and the formalism must be able to express this.

DEFINITION 19 (Passing of Pointers to Methods). *For methods $m, m' \in M(C)$, we define $PP(m, m')$ to be the number of invocations of m, where a pointer to m' is passed to m (via parameter). Such method invocations can be located in the body of any method in the system, not only in methods of the classes where m and m' are defined.*

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

From a practical perspective, the formalism can also be used simplify the construction of tools that automatically extract measures. Typically, measurement tools calculate the measures by performing queries on an abstract syntax tree or semantic graph representation of the system, as depicted in the upper part of Fig. 1.

Since measures can be expressed using the primitives of our formalism, we can also design a tool as depicted in the lower part of Fig. 1: The formalism is calculated by performing queries on a syntax tree (or, more conveniently, a semantic graph). The measures are then implemented relying only on the primitives of the formalism. This architecture has two advantages. First, new measures can be added with less effort, as most of them can be built easily from the existing primitives and no new queries have to be written. Second, the measurement tool can be more easily

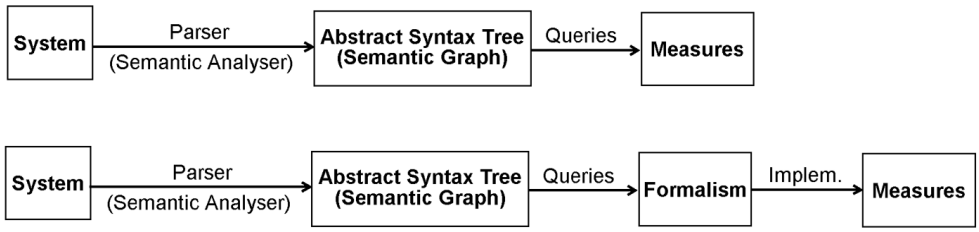


Fig. 1. Basic architecture of a measurement tool with and without our formalism.

adapted to accommodate other languages. If a different parser/semantic analyzer for a different language is used, only the (fewer) queries of the formalism have to be rewritten for the new language; the part of the tool that calculates the measures from the formalism primitives need not be changed.

4 Survey of Coupling Measurement Frameworks and Measures

In this section, we perform a comprehensive survey and critical review of existing frameworks and measures for coupling in object-oriented systems. In Section 4.1, we present existing frameworks for coupling and compare them. In Section 4.2, we present existing coupling measures, compare and discuss them, and analyze their mathematical properties.

4.1 Coupling Frameworks

Frameworks for coupling in object-oriented systems have been proposed by Eder et al. [20], Hitz and Montazeri [22], and Briand et al. [4]. In each framework different types of class, method, and object coupling are identified. We will present each of these frameworks in Section 4.1.1, and then compare and discuss them in Section 4.1.2.

4.1.1 Definition of the Frameworks

Framework by Eder et al. [20]. Eder et al. use the definition of coupling provided by Stevens et al. [35]. The authors identify the following types of relationships:

- *Interaction relationships between methods.* This type of relationship is caused by message passing.
- *Component relationships.* Each object has a unique identifier (the object identity). An object *o* may reference another object *p* using the identifier of object *p*. This introduces a component relationship between the classes of *o* and *p*.
- *Inheritance relationships between classes.*

From these relationships three dimensions of coupling are derived.

1) *Interaction coupling*: Two methods are interaction coupled if
   a) one method invokes the other, or
   b) they communicate via sharing of data.

   Interaction coupling between method *m* implemented in class *c* and method *m′* implemented in class *c′* contributes to interaction coupling between the classes *c* and *c′*.

2) *Component coupling*: Two classes *c* and *c′* are component coupled, if *c′* is the type of either
   a) an attribute of *c*, or
   b) an input or output parameter of a method of *c*, or
   c) a local variable of a method of *c*, or
   d) an input or output parameter of a method invoked within a method of *c*.

3) *Inheritance coupling*: two classes *c*, and *c′* are inheritance coupled, if one class is an ancestor of the other.

For each dimension of coupling, the authors identify different strengths of coupling (listed below from strongest to weakest).

Interaction coupling. The definition of interaction coupling is most similar to the original definition of coupling. Therefore, Eder et al. use the types of coupling proposed by Myers [31] and adapt these to object-oriented systems.

- *Content*. Method accesses implementation of another. Implementation here means the nonpublic part of the class interface. In C++, for instance, a method *m* may be declared "friend" of a class *c*. Method *m* can then invoke private methods of class *c*. Access to implementation constitutes a breach of the information hiding principle and is considered the worst type of coupling.
- *Common*. Methods communicate via unstructured, global, shared data space. The authors cannot give an example for an unstructured global shared data space, because there are no object-oriented languages which support them. Apparently, this type of coupling is only listed in order to be consistent with Myers' categories.
- *External*. Methods communicate via structured, global, shared data space (e.g., a public attribute of a class). Eder et al. find that this is a violation of the "locality principle of good software design," without specifying any further what they mean by that.
- *Control*. Methods communicate via parameter passing only, the called method controls the internal logic of the calling method. For instance, the called method may determine the future execution of the calling method. A change of the implementation of the called method will most likely affect the calling method (change dependencies).
- *Stamp*. Methods communicate via parameter passing only, the called method does not need all of the data it receives. This constitutes an avoidable dependency between methods. E.g., if the data structure of a parameter of a method is changed, possible effects of this change on the method have to be considered. If

the parameter is unused, a change of the parameter's data structure will not have any effects. The effort spent to discover that the change has no effects can be saved by avoiding stamp coupling.
- *Data.* Methods communicate via parameter passing only, the called method needs all the data it receives. This is the best type of coupling (besides no coupling at all), because it minimizes the change dependencies.
- *No direct coupling.* No direct interaction coupling between two methods occurs.

Eder et al. first consider only direct interaction coupling between two methods. Their definition is then expanded to indirect interaction coupling via transitive method invocations.

Component coupling. There are four degrees of component coupling between classes (listed below from strongest to weakest).

- *Hidden.* Component coupling does not manifest itself in code. For instance, if a class *c* contains a cascaded method invocation such as *a.m*1().*m*2(), the type of the object returned by *m*2 need not be explicit in the interface or body of class *c*. It can be found in the interface of the class of the object returned by method *m*1. That is, in order to detect occurrences of hidden coupling where class *c* is involved, we also have to look at the interfaces of other classes.
- *Scattered.* Component coupling manifests itself in the body of the class only (cases c) and d) in the above definition). Consequently, the body of the class has to be searched in order to detect occurrences of this type of coupling.
- *Specified.* Component coupling manifests itself in the interface (cases a) and b)). It is sufficient to search the interface of the class for occurrences of this type of coupling.
- *Nil.* No component coupling.

Inheritance coupling. There are four degrees of inheritance coupling (listed below from strongest to weakest).

- *Modification.* Inheriting class changes at least one inherited method in a manner that violates some predefined "good practice" rules. Eder et al. provide examples of such rules as "the signature of an inherited method *m* may only be changed by replacing the type of a parameter of *m*, say class *d*, with a descendent of class *d*," "an inherited method must not be deleted from the class interface," and "if a method is overridden, the overriding method must keep the same semantics as the overridden method." The predefined rules applied will, to a certain extent, depend on the used design methodology and programming language. When these rules involve the semantics of methods, they are subjective and not easily measured automatically from a static analysis of the design or source code documents. Modification coupling is the strongest type of inheritance coupling because information inherited from the parent class is modified or deleted in a manner which cannot be justified in the context of inheritance. Two types of modification coupling exist.
  1) *Signature modification*: Not only the implementation of at least one inherited method is changed, but the signature of the method is also changed.
  2) *Implementation modification*: The implementation of at least one inherited method is changed. This degree of coupling is weaker than the previous type because the signature of the method is not changed.
- *Refinement.* Inheriting class changes at least one inherited method but the change is made adhering to the predefined "good practice" rules. Refinement coupling is weaker than modification coupling because the inherited information is changed only according to the predefined rules. However, problems can still occur as a result of refinement coupling, e.g., changes to the signature of an inherited method will restrict the use of polymorphism even if the intended semantics of the method are not changed. Again, like modification coupling, there exist two different types of refinement coupling.
  1) *Signature refinement*: Not only the implementation of at least one inherited method is changed, but the signature of the method is also changed.
  2) *Implementation refinement*: The implementation of at least one inherited method is changed. This degree of coupling is weaker than the previous type because the signature of the method is not changed.
- *Extension.* Inheriting class changes neither the signature nor the body of any inherited method; only new methods and attributes are added.
- *Nil.* No inheritance relationship between two classes.

Framework by Hitz and Montazeri [23]. Hitz and Montazeri approach coupling by defining the state of an object (the value of its attributes at a given moment at run-time), and the state of an object's implementation (class interface and body at a given time in the development cycle). From these definitions, they derive two "levels" of coupling:

- *Class level coupling* (*CLC*). CLC represents the coupling resulting from state dependencies between two classes in a system during the development lifecycle.
- *Object level coupling* (*OLC*). OLC represents the coupling resulting from state dependencies between two objects during the run-time of a system.

According to Hitz and Montazeri, CLC is important when considering maintenance and change dependencies because changes in one class may lead to changes in other classes which use it. The authors also state that OLC is relevant for run-time oriented activities such as testing and debugging. For each of these levels of coupling, the authors identify a series of factors determining the strength of coupling.

1) *Class level coupling.* CLC can occur if a method of a class invokes a method or references an attribute of another class. In the following, let *cc* be the accessing class (client class), *sc* be the accessed class (server class). The factors determining the strength of CLC between *cc* and *sc* are:

- *Stability of sc* :
  - *sc is stable.* Interface or body of *sc* is unlikely to be changed (for instance due to changing requirements). Typically, basic types provided by the programming language, or classes imported from standard libraries are stable.
  - *sc is unstable.* To depend on an unstable server class is considered worse than to depend on a stable class because a change to *sc* means potential change to *cc*. Typically, problem domain classes are unstable. Two cases must be considered.
    1) only the body of *sc* is likely to be changed.
    2) the interface of *sc* may also be modified. This case is considered the more harmful modification.
- *Type of access*
  - "*Access to interface*": *cc* invokes a method of *sc*.
  - "*Access to implementation*": *cc* references an attribute of *sc*.

Access to implementation is considered stronger coupling as it constitutes a violation of the information hiding principle.

- *Scope of access.* Determines where *sc* is visible within the definition of *cc*. Within this scope, a change to *sc* may have an impact on *cc*. The larger the scope, the stronger the classes are coupled. The authors identify five cases which can be separated into two categories: 1) a reference to *sc* may occur in any method of *cc* and 2) a reference to *sc* can occur only through a particular method of *cc* (this becomes clear below).

  Category 1) is comprised of three cases:
  - *sc* is the type of an attribute of *cc*.
  - *sc* is an ancestor of *cc*.
  - *sc* is the type of a global variable.

  Category 2) is comprised of two cases:
  - *sc* is the type of a local variable of a method of *cc*.
  - *sc* is the type of a parameter of a method of *cc*.

2) *Object level coupling.* For the discussion of OLC (object level coupling), let o_{sc} be an object of type *sc*, o_{cc} an object of type *cc*. Three factors influence the strength of coupling between objects o_{sc} and o_{cc}:

- *Type of access.* o_{cc} accesses interface of o_{sc} or o_{cc} accesses implementation of o_{sc} (same distinction and implications for strengths of coupling as for CLC).
- *Scope of access.* The smaller the scope of access the weaker the coupling between the objects. For o_{cc} to be able to access o_{sc} and contribute to OLC object o_{sc} must be either (listed in increasing size of scope)
  1) a parameter of a method of o_{cc}
  2) a "nonnative" part of o_{cc}, that is, o_{sc} is not an object inherited from a superclass of o_{cc} nor is it encapsulated (aggregation) within o_{cc} nor is it a local variable to one of o_{cc} methods
  3) a global object.
- *Complexity of interface.* In the case that o_{cc} sends a message to o_{sc}, the number of parameters of the invoked method should be considered. The more parameters passed, the stronger the coupling between objects.

Framework by Briand et al. [4]. An earlier approach ([11], [12]) to measure coupling in object-based systems such as those implemented in Ada is adapted to C++ by expanding it to include inheritance and friendship relations between classes. In contrast to the two previous frameworks, this framework focuses solely on coupling relationships available during the high level design phase. The motivation behind this decision is that eliminating design flaws and errors early before they can propagate to subsequent phases can save substantial amounts of money. As a result of the decision to focus on early design information, this framework concentrates on coupling caused by interactions that occur between classes. Three different facets are identified that determine the kind of interaction:

1) *Type of interaction*. Determines the mechanism by which two classes are coupled. Different types of interaction are specified to determine if a particular type more accurately indicates fault likelihood.
   - *Class-attribute interaction*. There is a class-attribute interaction between classes *c* and *d*, if class *c* is the type of an attribute of class *d* (i.e., if aggregation occurs).
   - *Class-method interaction*. Let m_d be a method of class *d*. There is a class-method interaction between classes *c* and *d* if
     - class *c* is the type of a parameter of method m_d
     - class *c* is the return type of method m_d.
   - *Method-method interaction*. Let m_c be a method of class *c*, m_d be a method of class *d*. There is a method-method interaction between classes *c* and *d*, if
     - m_d directly invokes m_c
     - m_d receives via parameter a pointer to m_c thereby invoking m_c indirectly.
2) *Relationship*. In C++, two classes *c* and *d* can have one of three basic relationships:
   - *Inheritance.* Class *c* is an ancestor of class *d* or vice versa. This category is specified because the use of inheritance apparently contradicts the notion of minimizing coupling and should be considered separately. Coad and Yourdon proposed a design principle which recommends high coupling between a class and its parents, and low coupling between classes not related via inheritance [18], [19].
   - *Friendship.* Class *c* declares class *d* its friend which grants class *d* access to nonpublic elements of *c*. This category is specified because it breaks encapsulation and thus violates the information-hiding principle.
   - *Other.* No inheritance or friendship relationship between classes *c* and *d* .

3) *Locus*. The "locus of impact" of an interaction. If class *c* is involved in an interaction with another class, a distinction is made between
   - *export*. Class *c* is the used class (server) in the interaction, and
   - *import*. Class *c* is the using class (client) in the interaction.

The motivation for this distinction is to investigate whether direction is important for predicting the fault-proneness of a class.

In the definition of this framework, Briand et al. deliberately assign no strengths to the different kinds of interactions they propose. The authors state such strengths should be derived from empirical validation which can then be used to define measures on an interval or ratio scale. Briand et al. pose several hypotheses regarding these facets of coupling and investigate these empirically with respect to prediction of fault prone classes.

4.1.2 Discussion and Comparison of Frameworks

A precise comparison of the frameworks shows there are differences in the manner in which coupling is addressed. One reason for this is the different objectives of the frameworks. For example, Briand et al. examined only early design information to investigate potential early quality indicators while the other authors investigated information mainly available at low level design and implementation; hence differences are found in the mechanisms that constitute coupling. A second reason is that some of the issues dealt with by one set of authors are considered to be subjective and too difficult to measure automatically. For example, the stability of an individual class (addressed by Hitz and Montazeri) is not something which can be easily determined unless, say, all problem domain classes are classified as unstable. In the following, we discuss in detail the significant differences between the frameworks and what can be learned from these differences.

The mechanisms that constitute coupling. In Table 1, the mechanisms that constitute coupling according to each of the frameworks are presented. Each row represents one mechanism, an "X" indicates that the mechanism is covered by the framework in the respective column. The mechanisms are numbered for reference purposes.

There is some overlapping of the frameworks: Mechanisms 3, 5, and 6, are common to all three frameworks. Mechanisms 7 and 9 are common to two frameworks. All other mechanisms are unique to one of the frameworks.

Within the framework of Hitz and Montazeri, mechanisms 5, 6, 7, and 9 are possible 'scopes of access'. The scope of access determines the strength of the access itself, which can be accomplished by mechanisms 2 or 3 (the possible 'types of access').

Mechanism 9 (inheritance) is of a different nature than the other mechanisms. If two classes are connected via one of the mechanisms 1 to 8, then we have an actual usage relationship between the classes: one class uses the other. If two classes are connected via mechanism 9, i.e., one class is the ancestor of the other, then there can (and probably should), but need not be an actual usage relationship between the classes, and the interdependencies this entails do not necessarily exist.

The coupling mechanisms differ in the development phase in which they become applicable. For instance, attribute references and method invocations (mechanisms 2 and 3) are completely known only after implementation. In contrast, aggregation is visible in the class interface and is typically available before implementation starts. Coupling mechanisms that are applicable early in the development process are particularly interesting. If, for instance, they help in identifying fault-prone classes, this information could be used to select classes which are to undergo formal verification or inspection, to allocate the best people to the most fault-prone parts of the design, or to select the optimal design from a series of design alternatives before these are implemented. However, the later the development phase, the more detailed the description of the system under development, the more detailed the analysis of coupling.

Strength of coupling. The strength of coupling between two classes is determined by two aspects:

- the frequency of connections between the classes, and
- the types of connections between the classes.

TABLE 1
COMPARISON OF MECHANISMS THAT CONSTITUTE COUPLING

#	Mechanism	Eder *et al.* [20]	Hitz & Montazeri [22]	Briand *et al.* [4]
1	methods share data (public attributes etc.)	X		
2	method references attribute		X	
3	method invokes method	X	X	X
4	method receives pointer to method			X
5	class is type of a class' attribute (aggregation)	X	X	X
6	class is type of a method's parameter or return type	X	X	X
7	class is type of a method's local variable	X	X	
8	class is type of a parameter of a method invoked from within another method	X		
9	class is ancestor of another class	X	X	

The first aspect, how to count the frequency of connections between classes, has to be ultimately resolved when defining the measures. This aspect has not been addressed by any framework probably because the information required to make this decision is available only after the source code is developed. In Section 4.2, where the definitions of various proposed coupling measures are compared, it is shown that there are a number of different ways to count the frequency of connections between classes.

Different types of coupling have different strengths. Eder et al. and Hitz and Montazeri assign strengths to the types of coupling they identified by defining a partial order on the set of coupling types used in their frameworks. That is, for any two types of connections within each framework, they define if one is stronger than the other, if both have equal strength, or if their strengths are not comparable. It is important to note that the definition of such a partial order is to some degree subjective and requires empirical validation. Furthermore, the validity of a given order will clearly depend on the concrete measurement goal, i.e., different measurement goals can require different (partial) orders. For instance, assuming regression testing by means of structural testing based on control flow analysis is performed, we wish to estimate the effort to test a class based on the amount of import coupling of the class. We would be interested in method invocations because that influences the flow of control. We would not but be interested in references to attributes because these have no impact on the flow of control. If, on the other hand, we want to characterize the understandability of the class based on import coupling, direct references to attributes are likely to be equally important as method invocations.

Briand et al. define a set of measures which count for each interaction type of their framework the number of interactions a class has with other classes. Empirical validation is then conducted to evaluate their potential of identifying fault-prone classes. That is, strengths are empirically assigned to the different types of coupling.

Direction of coupling. The framework by Briand et al. explicitly distinguishes between import and export coupling. Consider two classes *c* and *d* being coupled through one of the mechanisms mentioned above. This introduces a client-server-relationship between the classes: the client class uses (imports services), the server class is being used (exports services). This distinction is important. A class which mainly imports services may be difficult to reuse in another context because it depends on many other classes. On the other hand, defects in a class which mainly exports services are particularly critical as they may propagate more easily to other parts of the system and are more difficult to isolate. We conclude that the direction of coupling measured directly influences the possible goals of measurement.

If two methods are coupled through "common" or "external" coupling according to the framework by Eder et al., we cannot make a distinction between client and server so both methods would be clients. Note that with pure object-oriented languages this would not occur. If the global data space is a variable whose type is a class, this class could be considered the server.

Direct and indirect coupling. Eder et al. derive "indirect interaction relationships between methods" from "direct interaction relationships" using the transitive closure of direct interaction relationships. This idea can be applied to all kinds of coupling. If a class c_1 uses a class c_2, which in turn uses a class c_3, class c_1 is indirectly coupled to c_3: a defect or modification in class c_3 may not only affect the directly coupled class c_2, but also the indirectly coupled class c_1. As an extreme case, consider a circular chain of coupled classes (class c_i uses class $c_i + 1$ for $i = 1, 2, ..., n - 1$, and class c_n uses c_1). Each class is directly coupled with two other classes (import and export coupling). However, each class in the chain indirectly uses and is being used by every other class.

Briand et al. based their framework on the work described in [12]. In [12], high-level design measures for coupling and cohesion in object-based systems were defined and validated with respect to their potential of identifying fault-prone modules. The coupling measures included measures for direct and indirect coupling. The measures for direct coupling were found to be useful predictors, but not those for indirect coupling. Because their framework has primarily been defined to derive coupling measures for the identification of fault-prone classes, Briand et al. did not include the distinction between direct and indirect coupling in their framework in [4].

Stability of server class. This point is unique to the framework by Hitz and Montazeri. Using a stable class is better than using an unstable class, because modifications which could ripple through the system are less likely to occur. "Stability of the server class" could, for instance, be used to distinguish between classes imported from standard libraries (which usually are not being modified and thus are stable), and problem domain classes (which are unstable). Note that stability of a server class is a subjective concept which is difficult to measure automatically in a manner other than that suggested above.

Inheritance. An aspect unique to object-oriented systems is how inheritance influences coupling. This aspect is addressed in some detail by Eder et al. and Briand et al. For the discussion, let us consider the case where a method of one class invokes a method of another. The question is: since both the invoking and the invoked method can be either declared or implemented in their classes, how does this affect coupling between the classes?

Consider the example in Fig. 2. There are four points worth noting here:

1) If a method invokes another method, the invoking method contributes to import coupling of its class from other classes. What if the invoking method is inherited? In Fig. 2, method *mc*1 in class *c*1 invokes *md*1 in *d*1. Method *mc*1 is also inherited to class *c*2. Should this contribute to coupling between classes *c*2 and *d*1? And what if an inherited method is redefined, as in the case of *mc*2 of *c*2. Eder et al. state that coupling between classes requires the invoking method to be implemented in the client class. Applied to our example, it follows that *mc*1 declared in *c*2 does not contribute to coupling of *c*2, whereas *mc*2, which is implemented in *c*2, does.

```
class c1 {
     d1 *o1;
   public:
     void mc1();
     void mc2();
};
void c1::mc1() { o1->md1(); }
void c1::mc2() { o1->md2(); }


class c2 : public c1 {
     d2 *o2;
   public:
     void mc2(); /* redefined */
     void mc3();
};
void c2::mc2() { o2->md2(); }
void c2::mc3() {
       mc1(); o2->md1();
       o2->md2();
}
```

```
class d1 {
   public:
     virtual void md1();
     virtual void md2();
};
void d1::md1() { ... }
void d1::md2() { ... }

class d2 : public d1 {
   public:
     void md2(); /* override */
     void md3();
};
void d2::md2() { ... }
void d2::md3() { ... }

class d3 : public d1 {
   public:
     void md3();
   };
void d3::md3() { ... }
```

Fig. 2. Example inheritance hierarchy.

2) If we agree that an inherited method does not contribute to import coupling of the inheriting class, then what about an invocation of an inherited method? For instance, *mc*3 of *c*2 invokes *mc*1 implemented in *c*1. The authors of all frameworks discussed here find that invoking an inherited method is special. It contributes to coupling, and has to be distinguished from invoking a method of an unrelated class. This type of coupling is commonly referred to as "inheritance-based" or "inheritance-related" coupling.
3) Due to polymorphism, one method invocation can actually access a variety of methods implemented in different classes. Consider method *mc*2 of *c*1 invoking *md*2 of *d*1. This clearly contributes to coupling between *c*1 and *d*1. However, the dynamic type of the object pointed to by o1 may be any descendent class of *d*1. Does *mc*2 therefore also contribute to coupling between *c*1 and *d*2 or *d*3? Eder et al. state that interaction coupling requires the invoked method to be implemented in the server class. Applied to our example, method *mc*2 of class *c*1 contributes to coupling between *c*1 and *d*1, and between *c*1 and *d*2. It does not couple *c*1 to class *d*3, because method *md*2 is only declared in class *d*3. The rationale behind this decision is that within the framework of Eder et al., all possible relationships, and thus all possible dependencies between methods should be accounted for.
4) In 2) we discussed the case that a class invokes a method it inherited from its parent class. Let us now consider the case that the client and server classes are not related through inheritance. The client class can invoke a method which the server class has inherited. For instance, method *mc*3 of class *c*2 invokes method *mc*1 of *d*2. Class *d*2 has inherited *mc*1 from *d*1. Should this therefore contribute to coupling between *c*2 and *d*1? And what about the same method *mc*3 invoking method *md*2 of *d*2? Class *d*2 inherits but overrides method *md*2 of *d*1. Again, we can apply the principle of Eder et al. that the invoked method has to be implemented in the server class. It follows, that the invocation of *md*2 contributes to coupling between *c*2 and *d*2 (because *md*2 is implemented in *d*2). The invocation of *md*1 contributes to coupling between *c*2 and *d*1 (because *md*1 is only declared in *d*2, but implemented in *d*1).

Class level coupling vs. object level coupling. A distinction unique to the framework by Hitz and Montazeri [22] is that between class level coupling (CLC) and object level coupling (OLC). The types of coupling dealt with in the other two frameworks would be considered class level coupling in the framework by Hitz and Montazeri, i.e., usage dependencies between classes that can be determined from a static analysis of the design documents or source code only. OLC, on the other hand, depends on the concrete object-structure at run-time, which in turn is determined by the actual input data to the system. That is, OLC is a function of both the design or source code and some input data at run-time.

4.1.3 Summary

To summarize, the following about coupling in object-oriented systems is noted:

- there are different types of coupling among classes, methods, attributes
- classes and methods can be coupled more or less strongly, depending on
  1) the type of connections between them
  2) the frequency of connections between them
- a distinction can be made between import and export coupling (client-server relationships)
- both direct and indirect coupling may be relevant
- the server class can be stable or unstable
- the effect of inheritance on coupling has to be considered.

From this list we can see that there exists a variety of decisions to be made during the definition of a coupling 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. A second observation is that because the different aspects of coupling are independent of each other, a large number of coupling measures could be defined—this defines the problem space for coupling in object-oriented systems. In the following section, a review of object-oriented coupling measures in the software engineering literature is presented. Discussion of how existing measures address the different aspects of coupling takes place and insight is provided into how complete the overall problem space for coupling is covered by these measures.

4.2 Coupling Measures

Coupling measures have been proposed by Chidamber and Kemerer [14], [15], Li and Henry [26], Martin [29], Abreu et al. [1], Lee et al. [28], and Briand et al. [4]. In Section 4.2.1, we present the original definitions of these measures, point out ambiguities in their definitions, and rewrite the definitions using the formalism presented above. The measures are then compared and discussed in Section 4.2.2. In Section 4.2.3, we analyze their mathematical properties.

4.2.1 Definition of the Measures

***Coupling between objects* (CBO)** [14], [15]. Measure CBO is defined in [14] as follows: "CBO for a class is a count of the number of noninheritance related couples with other classes." An object of a class is coupled to another, if methods of one class use methods or attributes of the other. In [15], a revised definition is proposed: "CBO for a class is a count of the number of other classes to which it is coupled." A footnote in another place says that "this includes coupling due to inheritance."

"Noninheritance coupling" and "coupling due to inheritance" refer to the case that a method m of a class c uses a method or attribute of an ancestor class of c. From this, we infer that the calling method m of class c has to be implemented at c (because an inherited method is considered a method of "another class"). The authors do not specify how polymorphism, method overriding and other issues raised in Section 4.1.2 affect coupling. We define two versions of CBO:

DEFINITION 20 (Measures CBO and CBO').

$$CBO(c) = \left|\{d \in C - \{c\} \,\middle|\, uses(c, d) \vee uses(d, c)\}\right|$$

$$CBO'(c) = \left|\{d \in C - (\{c\} \cup Ancestors(C)) \,\middle|\, uses(c, d) \vee uses(d, c)\}\right|$$

Class c is coupled to class d if it uses d or is being used by d. The definition of predicate $uses(c, d)$ demands that both the using method of class d and the used method or attribute of class d are implemented at their classes. Polymorphism is accounted for. Other interpretations are possible, the predicate *uses* would have to be defined appropriately. For instance, if we wanted to consider static method invocations only, we would define:

$$uses(c, d) \Leftrightarrow (\exists m \in M_I(c): \exists m' \in M_I(d): m' \in SIM(m))$$
$$\vee (\exists m \in M_I(c): \exists a \in A_I(d): a \in AR(m))$$

***Response for class* (RFC)** [14], [15]. Original definition [15]: "$RFC = |RS|$ where RS is the response set for the class. The response set can be expressed as $RS = \{M\} \cup_{all\ i} \{R_i\}$, where $\{R_i\}$ is the set of methods called by method i, and $\{M\}$ is the set of all methods in the class. The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class."

The last sentence in this definition indicates that the sets R_i do not only include the methods directly invoked by method i, but also the methods called by these methods and so on. In a footnote, the authors say that "membership to the response set is defined only up to the first level of nesting of methods called due to practical considerations involved in collection of the metric." This restriction has not been made in [14]. Does the set M of all methods in the class include inherited methods? The answer is probably yes, because these inherited methods may be invoked in response to a message to that object.

We define the following RFC measures:

DEFINITION 21 (Measure RFC_α). *Let* $R_0(c) = M(c)$, *and let* $R_{i+1}(c) = \bigcup_{m \in R_i(c)} PIM(m)$ *be the set of methods polymorphistically invoked by methods in* $R_i(c)$. *Then*

$$RFC_\alpha(c) = \left|\bigcup_{i=0}^{\alpha} R_i(c)\right|, \quad \text{for } \alpha = 1, 2, 3, \ldots,$$

RFC_α takes into account levels of nested method invocations. This measure has been suggested by Churcher and Shepperd [17]. We can now define the measures proposed by Chidamber and Kemerer as special cases of RFC_α:

DEFINITION 22 (Measures RFC and RFC').

$$RFC'(c) = RFC_\infty(c), \text{ and}$$
$$RFC(c) = RFC_1(c).$$

***Message passing coupling* (MPC)** [26]. Measure MPC is defined as the "number of send statements defined in a class." We further quote [26]: "The number of send statements sent out from a class may indicate how dependent the implementation of the local methods is on the methods in other classes." This indicates that MPC only counts invocations of methods of other classes, not invocations of the class' own methods. Also, only send statements in "local methods" are counted. Li and Henry write: "The local methods of a class constitute the interface increment." Inherited methods are not part of the interface increment; therefore send statements in inherited methods are not counted. If a class overrides an inherited method, it is not strictly an interface increment. Should this method still be excluded? Since invocations from inherited methods are not counted, should we count invocations of inherited methods? We provide a formal definition of MPC as follows:

DEFINITION 23 (Measure MPC).

$$MPC(c) = \sum_{m \in M_I(c)} \sum_{m' \in SIM(m) - M_I(c)} NSI(m, m').$$

It is the number of static invocations of methods not implemented in c by methods implemented in c. Only static

invocations are counted, so that this count reflects the number of send statements.

***Data abstraction coupling* (DAC)** [26]. Measure DAC has been defined as "the number of abstract data types (ADTs) defined in a class." In this context, an ADT is a class in the system. An ADT is defined in a class *c*, if it is the type of an attribute of class *c*. We further quote [26]: "The number of variables (remark: attribute in our terminology) having an ADT type may indicate the number of data structures dependent on the definitions of other classes." This raises the question if DAC is supposed to count the number of ADTs defined in a class, or the number of attributes having an ADT? Consider this example:

```
class A {/* ... */};
class B {A a1, a2;};
```

The number of ADTs defined in B is 1, the number of attributes having an ADT is 2. And again, we have to decide if inherited attributes should be included in the count or not. We define two versions of DAC:

DEFINITION 24 (Measure DAC).

$$DAC(c) = \left|\{a \mid a \in A_I(c) \wedge T(a) \in C\}\right|$$
$$DAC'(c) = \left|\{T(a) \mid a \in A_I(c) \wedge T(a) \in C\}\right| .$$

DAC is the number of not inherited attributes that have a class as their type. The number of the classes used as types for attributes is counted by DAC'.

Ce *and* Ca (*efferent and afferent coupling*) [29]. For the following definition, a category is a set of classes that belong together in the sense that they achieve some common goal. The original definition of the Ce and Ca is:

- Ca: The number of classes outside this category that depend upon classes within this category.
- Ce: The number of classes inside this category that depend upon classes outside this category.

Martin does not specify exactly what constitutes dependencies between classes. We may, for instance, adapt the idea of Chidamber and Kemerer that class *c* depends on class *d* if a method of class *c* uses a method or attribute of class *d*. However, we will not provide formal definitions for Ce and Ca.

***Coupling factor* (COF)** [1]. In the following, the system consists of classes c_1, c_2, ..., c_{TC}, where *TC* is the total number of classes in the system. The function *isclient*(*cc*, *cs*) is 1, if class *cc* is not a descendent of class *cs*, and $cc \neq cs$ and *cc* references a method or attribute of class *cs*. Otherwise, *isclient*(*cc*, *cs*) is 0. The original definition of *COF* is:

$$COF(S) = \frac{\sum_{i=1}^{TC}\sum_{j=1}^{TC} isclient(c_i, c_j)}{TC^2 - TC - \left(2\sum_{i=1}^{TC}\left|Descendents(c_i)\right|\right)}$$

The numerator of COF is the actual number of client-server relationships between classes that are not related via inheritance. The denominator is the maximum possible number of such client-server-relationships. COF is normalized to range between 0 and 1, in order to allow for comparisons between systems of different size.

The authors do not specify if and how they account for polymorphism, method overriding and other issues raised in Section 4.1.2. Using our formalism, we define COF as follows:

DEFINITION 25 (Measure COF).

$$COF(C) = \frac{\sum_{c\in C}\left|\{d \mid d \in C - (\{c\} \cup Ancestors(c)) \wedge uses(c,d)\}\right|}{|C|^2 - |C| - \left(2\sum_{c\in C}\left|Descendents(c)\right|\right)}$$

***Information-flow-based coupling* (ICP)** [28]. The original definitions of the ICP measures use a formalism that would be rather lengthy to reproduce here. Using our formalism, these measures are defined as follows:

DEFINITION 26 (Measure ICP).

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

$ICP^c(m)$ counts for method m of class c, the number of polymorphistically invoked methods of other classes, weighted by the number of parameters of the invoked method. This count is scaled up to the classes and subsystems:

$$ICP(c) = \sum_{m\in M_I(c)} ICP^c(m)$$
$$ICP(SS) = \sum_{c\in SS} ICP(c)$$

From these measures, Lee et al. derive two more sets of measures which measure separately coupling to ancestor classes (inheritance-based coupling) and coupling to unrelated classes (noninheritance-based coupling).

DEFINITION 27 (Measures NIH-ICP AND IH-ICP). *The set of noninheritance-based measures is:*

$$NIH{-}ICP^c(m) = \sum_{m'\in R} (1 + |Par(m')|)\cdot NPI(m,m'),$$

where

$$R = PIM(m) \cap \left(\bigcup_{c'\in Ancestors(c)} M(c')\right),$$

$$NIH{-}ICP(c) = \sum_{m\in M_I(c)} NIH{-}ICP^c(m),$$

and

$$NIH{-}ICP(SS) = \sum_{c\in SS} NIH{-}ICP(c).$$

The set of inheritance-based measures is:

$$IH{-}ICP^c(m) = \sum_{m'\in R} (1 + |Par(m')|)\cdot NPI(m,m'),$$

where

$$R = PIM(m) \cap \left(\bigcup_{c'\in C-(\{c\}\cup Ancestors(c))} M(c')\right),$$

$$IH{-}ICP(c) = \sum_{m\in M_I(c)} IH{-}ICP^c(m),$$

and

$$IH{-}ICP(SS) = \sum_{c \in SS} IH{-}ICP(c).$$

ICP is simply the sum of IH-ICP and NIH-ICP. For instance, ICP(c) = IH-ICP(c) + NIH-ICP(c).

Suite of measures by Briand et al. [4]. Briand et al. implemented their framework for coupling in object-oriented systems (see Section 4.1.1). They defined measures which count for each class.

- the number of class-attribute / class-method / method-method interactions
- originating from / directed at
- ancestor / friend / other classes.

There is a class-attribute (CA-)interaction from class *c* to class *d*, if an attribute of class *c* is of type class *d*. The number of class-attribute interactions from class *c* to class *d* can formally be expressed as

$$CA(c, d) = \left|\{a \mid a \in A_I(c') \wedge T(a) = d\}\right| .$$

There is a class-method (CM-) interaction from class *c* to class *d*, if a newly defined method of class *c* has a parameter of type class *d*. The number of CM-interactions from class *c* to class *d* can formally be expressed as

$$CM(c, d) = \sum_{m \in M_{NEW}(c)} \left|\{a \mid a \in Par(m) \wedge T(a) = d\}\right|.$$

There is a method-method (MM-)interaction from class *c* to class *d*, if a method implemented at class *c* statically invokes a method of class *d* (newly defined or overriding), or receives a pointer to such a method. The number of method-method interactions from class *c* to class *d* can formally be expressed as:

$$MM(c, d) = \sum_{m \in M_I(c)} \sum_{m' \in M_{NEW}(d) \cup M_{OVR}(d)} (NSI(m, m') + PP(m, m')).$$

DEFINITION 28 (Class-Attribute Interaction Measures).

$$IFCAIC(c) = \sum_{d \in Friends^{-1}(c)} CA(c, d)$$

counts all CA-interactions from class c to inverse friends of c,

$$ACAIC(c) = \sum_{d \in Ancestors(c)} CA(c, d)$$

counts all CA-interactions from class c to ancestors of class c,

$$OCAIC(c) = \sum_{d \in Others(c) \cup Friends(c)} CA(c, d)$$

counts CA-interactions from class c to classes that are not ancestors or inverse friends of class c. Others is defined as

$$Others(c) = C - (Ancestors(c) \cup Descendents(c) \cup Friends(c) \cup Friends^{-1}(c) \cup \{c\}).$$

$$FCAEC(c) = \sum_{d \in Friends(c)} CA(d, c)$$

counts all CA-interactions from friends of class c to class c,

$$DCAEC(c) = \sum_{d \in Descendents(c)} CA(d, c)$$

counts all CA-interactions from descendents of class c to class c,

$$OCAEC(c) = \sum_{d \in Others(c) \cup Friends^{-1}(c)} CA(d, c)$$

counts all CA-interactions to class c from classes that are not friends or descendents of class c.

The definitions of the class-method interaction measures IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, and OCMEC, and the method-method interaction measures IFMMIC, AMMIC, OMMIC, FMMEC, DMMEC, and OMMEC follow the same template. Instead of CA-interactions, these measures count CM- and MM-interactions, respectively. For instance,

$$IFCMIC(c) = \sum_{d \in Friends^{-1}(c)} CM(c, d)$$

and

$$OMMEC(c) = \sum_{d \in Others(c) \cup Friends^{-1}(c)} MM(d, c).$$

The full definitions of these and all other measures introduced in this section are summarized in Table 13 of Appendix C.

4.2.2 Discussion and Comparison of the Measures

In this section, we discuss and compare the coupling measures introduced in the previous section. In particular, we will analyze how the existing coupling measures address the issues highlighted in the comparison of frameworks in Section 4.1.2.

Types of coupling. Many of the coupling measures are based on method invocations and attributes references. All versions of RFC, MPC, and the ICP family by Lee et al. are based solely on method invocations. The "method-method interaction" measures OMMIC, IFMMIC, AMMIC and OMMEC, FMMEC, and DMMEC count method invocations plus occurrences where a method is passed a pointer to another method. And measures CBO and COF include references to both methods and attributes.

DAC, DAC' and the "class-attribute interaction" measures IFCAIC, ACAIC, OCAIC and FCAEC, DCAEC, and OCAEC are measures which take into account aggregation and could be classified as "component coupling" according to the framework by Eder et al. The "class-method interaction" measures IFCMIC, ACMIC, OCMIC and FCMEC, DCMEC, and OCMEC in the suite of measures by Briand et al. also measure "component coupling." These measures count occurrences where a parameter of a method has another class as its type.

Comparison of the types of coupling used by the coupling measures to those introduced in the coupling frameworks in Table 1 shows that all types of coupling used by the measures are present in at least one framework. In contrast, however, there are types of coupling in Table 1 for which there are no measures defined, namely, type #7 (a method of a class *c* has a local variable of class *d*), and type #8 (a method of a class *c* invokes a method while passing an

object of type class *d*). These types of coupling may be less interesting because they typically are available only after implementation. In conclusion, there are types of coupling in object-oriented systems which have not yet been considered—empirical studies are required to investigate the usefulness of these types.

Strength of coupling. In this section, we first examine how different measures account for different strengths of coupling. We then examine how the measures deal with the frequency of coupling connections.

Measures CBO and COF do not distinguish between method invocations and attribute references; both types of coupling are treated as one and the same. The "method-method interaction" measures by Briand et al. do not distinguish between method invocation and passing a pointer to a method *m′* as a parameter to some other method m (because m can then invoke *m′*). All other measures focus only on one type of coupling. Incorporating more than one type of coupling into a single measure may be questionable and needs to be clearly justified: it requires relative strengths to be assigned to the different types of coupling under consideration (e.g., the measures just mentioned treat different types of connections as one and the same). Such an assignment is subjective and makes empirical validation difficult. Questions will arise such as "What type of coupling contributed how much to the measurement values?" To answer these questions, the types of coupling have to be measured separately. So, unless there is a clear justification, it is strongly recommended not to combine different types of coupling. In addition, the assignment of relative strengths of coupling will depend on the measurement goal and other environmental factors (e.g., design methodology, programming language). Assigning strengths to the various types of coupling should be part of a prediction model for some external attribute (e.g., a model that predicts fault-proneness of a class from different types of coupling), but not part of the definition of a measure.

"Connection" is a generic term defined as an occurrence of a given type of coupling (e.g., a method invocation or an attribute having a class as its type). The measures CBO and COF take a binary approach to coupling between classes: two classes are either coupled or not and the number of such "class couples" is counted. However, this approach does not make use of all the information available. For instance, measure CBO has been proposed by Chidamber and Kemerer as an indicator for maintainability, testability and reusability of a class [15]. However, the maintainability and testability of the class is also likely to be influenced by the frequency of connections between coupled classes and not only by the number of classes to which it is coupled. For example, a class *c* which is loosely coupled to (i.e., has few connections to) five other classes may be easier to maintain or test than a class *d* which is strongly coupled (i.e., has many connections) to only two other classes. All other measures include individual connections between classes. RFC counts the number of methods invoked by a class. MPC and the "method-method interaction" measures by Briand et al. count the number of method invocations. The ICP measures also take the number of parameters passed to each method into account (within the framework by Hitz and Montazeri this is the factor "Complexity of interface" for object level coupling). DAC and the measures for class-attribute and class-method interaction by Briand et al. count the number of attributes and parameters having a class type. DAC' counts the number of classes used as a type for an attribute.

An important difference is between the "number of methods invoked" and the "number of method invocations." If the same method is invoked more than once and each invocation is counted separately, a distorted value for the measure can arise. Consider two extreme cases in an example given by Hitz and Montazeri [23]:

- Class c_1 invokes a method *m* 10 times.
- Class c_2 invokes 10 different methods of 10 different classes, once each.

For measure MPC (and other measures counting method invocations), $MPC(c_1) = MPC(c_2) = 10$. In reality, however, the coupling of class c_2 could be considered worse than that of c_1 because c_2 is coupled to 10 different classes whereas c_1 is coupled to only one.

Similar distorted values arise for attributes. Contrast measure DAC which counts the number of attributes having a class as its type with DAC´ which counts the number of classes used as types of attributes. If a class c_1 has 10 attributes of type class c_2, $DAC(c_1) = 10$ whereas $DAC'(c_1) = 1$.

At this point, it can be concluded that there are several ways to take into account the frequencies of connections between classes. They all have their strengths and weaknesses, and they all can be justified by an empirical relational system. That is, there is not "one right way" to count frequencies of connections. How to count them must be decided with respect to a given measurement goal.

Import and export coupling. CBO makes no distinction between import and export coupling: two classes are coupled if one uses the other or vice versa. COF distinguishes the cases where a class c_1 uses a class c_2, and class c_2 uses class c_1. In the case where both classes use each other both relationships will be counted separately. The suite by Briand et al. provides separate measures for import and export coupling (e.g., OMMEC and OMMIC). All other measures only consider import coupling (i.e., the role of the class as client).

Direct and indirect coupling. Most of the coupling measures consider direct coupling only. RFC´ is the number of methods that can possibly be invoked by sending a message to a class *c*. This includes methods of *c*, methods invoked by the methods of *c*, the methods these in turn invoke, etc. In that sense, indirect coupling is accounted for. RFC_α counts such nested method invocations up to a specified level α.

We can easily derive new measures that account for indirect coupling from measures that do not (if it appears sensible to do so). Direct coupling describes a relation on a set of elements (e.g., a relation "invokes" on the set of all methods of the system, or a relation "uses" on the set of all classes of the system). To account for indirect coupling, we need only use the transitive closure of that relation.

Stability of server class. This aspect is not addressed by any of the proposed coupling measures. A possible application would be to distinguish between coupling among prob-

lem domain classes and import coupling from classes taken from standard libraries (or any other classes that are not subject to development or change in the ongoing project).

A pragmatic method of identifying the stability of server classes without defining new measures is to use an existing import coupling measure and measure, for a given class, its import coupling from problem domain classes and library classes separately. Thus, it can be determined to what extent the class relies on problem domain classes and library classes.

Inheritance. The distinction between inheritance-based and noninheritance based coupling can be found frequently in the literature. Inheritance-based coupling refers to coupling between a class and its ancestors. Noninheritance-based coupling is coupling between two classes with no inheritance relationship between them. For instance, the design principle by Coad and Yourdon suggests to maximize inheritance-based coupling and minimize noninheritance-based coupling [18], [19]. Excluding inheritance-based coupling corresponds with the point of view that the inheriting class "has" the attributes and methods it inherits, and using an inherited method or attribute is not equivalent to coupling with another class. Including inheritance-based coupling corresponds with the point of view that the inheriting class does not "have" the inherited attributes and methods. This view is supported by the fact that some programming languages restrict access to inherited methods and attributes (e.g., private methods and attributes in a C++ class cannot be used by its children classes). Empirical studies are required to realize if and how inheritance-based and noninheritance-based coupling should be distinguished. It is also conceivable that the relative importance of both types of coupling depends on the respective measurement goal: what is empirically justified in one situation may not apply in other situations.

CBO' measures "noninheritance-based coupling" [14] whereas CBO explicitly includes "coupling due to inheritance" [15]. Chidamber and Kemerer do not explain why they first excluded inheritance-based coupling and why they introduced it later. Similarly, the definition of COF excludes inheritance-based coupling without any explanation [1].

Lee et al. acknowledge the need to differentiate between inheritance-based and noninheritance-based coupling by proposing corresponding measures: NIH-ICP counts noninheritance-based coupling only, IH-ICP counts inheritance-based coupling only. ICP is the sum of IH-ICP and NIH-ICP, thus treats both types of coupling equal. The suite of measures by Briand et al. also provides measures which count inheritance-based and noninheritance-based coupling separately. Unlike Lee et al., they do not define a single measure which counts both types of coupling (cf. the above discussion on types of coupling). All other measures do not address inheritance. The definitions of these measures are therefore ambiguous with that respect.

For most measures, the other inheritance issues introduced in Section 4.1.2 (how to account for polymorphism, overriding of methods, virtual methods etc.) are not considered in the original definitions. Reference [20] is the only publication we are aware of that describes these issues in detail.

Lee et al. emphasize that their ICP measures, which are based on method invocations, take polymorphism into account [28]. Their measures aim at measuring the amount of information flow between methods. Information flow occurs between a method m and any method m' that is possibly invoked by m which includes methods $m' \in PIM(m)$. The measures for method-method-interaction by Briand et al. are counts of static method invocations only [4].

Class level coupling vs. object level coupling. All measures are defined to measure class-level coupling in the framework of Hitz and Montazeri [22], i.e., they count static usage dependencies between classes. No measure of object level coupling has been proposed. This may reflect the practical difficulty of measuring coupling between individual objects. For instance, for object level coupling it is not sufficient to know how many invocations from one method to another there are, but how often these will be executed at run-time. One way to measure coupling between objects would be to instrument the source code to log all occurrences of object instantiations, deletions, method invocations, and direct references to attributes while the system executes. However, this kind of measurement can only be performed very late in the development process, e.g., in parallel to or after testing.

4.2.3 Mathematical Properties of the Measures

In this section, we investigate the properties of the coupling measures presented in Section 4.2.1. In particular, we look at five mathematical properties proposed by Briand et al. [13]. The motivation behind defining such properties is that a measure must be supported by some underlying theory of the internal quality attribute it is purported to measure—if not, how do we know a measure is measuring the intended attribute (also referred to as construct validity)? The five coupling properties defined by Briand et al. are one of the more recent proposals to characterize coupling in a reasonably intuitive manner. These properties are considered to be necessary, but not sufficient, to justify a coupling measure.

The coupling properties were originally defined using a generic, mathematical framework to model a software system. For the purpose of validating coupling measures for object-oriented systems, we adapt this framework to model an object-oriented system as defined by the formalism in Section 3.

In the following discussion let *Coupling* be a candidate measure for coupling of a class or an object-oriented system. Relationships capture the connections between classes the respective coupling measure is focused on. As the coupling measure can measure import or export coupling (or both), $OuterR(c)$ will denote the relevant set of relationships from or to class c (or both). We define $InterR(C) = \cup_{c \in C} OuterR(c)$ to be the set of interclass relationships in system C.

The five proposed coupling properties are:

Coupling.1: Nonnegativity. The coupling [of a class c | of an object-oriented system C] is nonnegative:

$$[Coupling(c) \geq 0 \mid Coupling(C) \geq 0]$$

Coupling.2: Null value. The coupling [of a class c | of an object-oriented system C] is null if [$OuterR(c)$ | $InterR(C)$] is empty:

$$[OuterR(c) = \phi \Rightarrow Coupling(c) = 0$$
$$\mid InterR(C) = \phi \Rightarrow Coupling(C) = 0]$$

Coupling.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 $OuterR(c) \subseteq OuterR(c')$, i.e., we added some relationships to c. Let C' be the object-oriented system which is identical to C except that class c is replaced by class c'. Then

$$[Coupling(c) \leq Coupling(c') \mid Coupling(C) \leq Coupling(C')]$$

Coupling.4: Merging of 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'. Then

$$[Coupling(c_1) + Coupling(c_2) \geq Coupling(c')$$
$$\mid Coupling(C) \geq Coupling(C')]$$

Coupling.5: 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

$$[Coupling(c_1) + Coupling(c_2) = Coupling(c')$$
$$\mid Coupling(C) = Coupling(C')]$$

Coupling.3 specifies that if a relationship is added to the system, coupling must not decrease. Coupling.4 specifies that merging two classes must not increase coupling because relationships disappear (namely those between the classes that have been merged). Coupling.5 specifies that merging two unconnected classes must not affect coupling at all.

In the following we discuss which measures violate one or more of the coupling properties above.

- RFC α and its special cases RFC and RFC' do not have a null value (Coupling.2). If c is a class with five methods which do not invoke any other methods, we have $RFC_\alpha(c) = 5$, even though $OuterR(c) = 0$.
- Measures DAC and DAC' also do not have a null value (Coupling.2). If class c has an attribute of type class c (or, more realistically, an attribute of type "pointer to class c"), then $DAC(c) > 0$ and $DAC'(c) > 0$, even though $OuterR(c) = 0$.
- COF violates Coupling.4 and Coupling.5. Consider the example systems in Fig. 3. Boxes are classes, arrows from, e.g., class c to class e, indicate that class c uses class e. For system C, we have COF(C) = 4/20 = 0.2. In system C', classes c and d have been merged to form a new class c'. We have $COF(C') = 3/12 = 0.25$, and thus COF(C) < COF(C'). Therefore, Coupling.4 and Coupling.5 are violated. This is due to the fact that COF has been normalized to range between [0, 1].
- CBO and CBO' do not fulfill Coupling.5. Again, consider the example systems in Fig. 3. For system C, we have $CBO(c) + CBO(d) = 2 + 2 = 4$. In system C' it is $CBO(c') = 3$. Thus Coupling.5 is violated.
- Likewise, measure DAC´ also violates Coupling.5. To see this, we reinterpret the meaning of the arrows in Fig. 3: an arrow from class c to class e now indicates that class c has one or more attributes of type class e. Again, we have $DAC'(c) = DAC'(d) = 2$ for system C, but $DAC'(c') = 3$ for system C', and Coupling.5 is violated. Measure DAC, on the other hand, fulfills Coupling.5. We always have $DAC(c) + DAC(d) = DAC(c')$, because this measure counts the number of attributes having a class as their type.
- Likewise, RFC_α and its special cases RFC and RFC´ violate Coupling.5. If we merge two classes c and d to form a new class c', the response set of c', $RS_{c'}$ is the union of the response sets RS_c and RS_d of classes c and d: $RS_{c'} = RS_c \cup RS_d$ (the response sets include the levels of nested method invocations as prescribed by parameter). If $RS_c \cap RS_d \neq \phi$, then $|RS_c| + |RS_d| > |RS_{c'}|$, and thus $RFC_\alpha(c) + RFC_\alpha(d) > RFC_\alpha(c')$, i.e., Coupling.5 is violated.

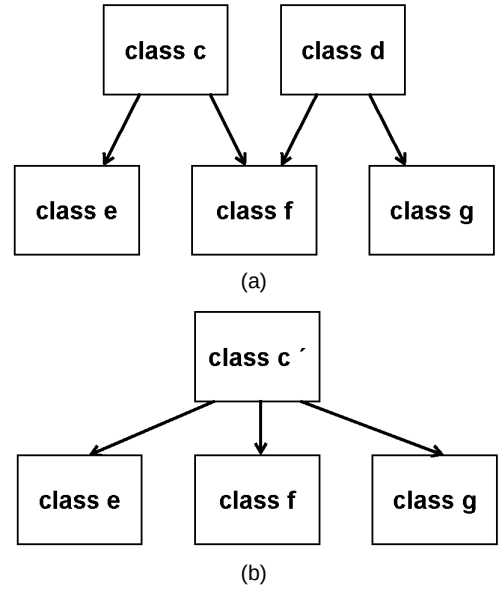


Fig. 3. Counterexamples for COF, CBO, CBO′, and DAC′. (a) system C; (b) system C'.

There is a pattern visible concerning the violations of property Coupling.5. Measures CBO, CBO', the RFC measures, and DAC´ all have in common that multiple connections to the same method or class are counted as one. If two unconnected classes c and d use a third class f in common, and we merge classes c and d to form a new class c', then the previously two connections to class f are only counted as one connection from new class c' to class f. As a result, the coupling of the new class c' is lower than the sum the of the coupling of classes c' and d. Thus, property Coupling.5 only holds for measures which count individual connections.

The ICP "information-flow-based" measures fulfill all five coupling properties, but show a special behavior. They measure the amount of information flowing in to and out from the class via parameters through method invocation,

i.e., the measures sum the number of parameters (plus one) passed at each method invocation. Consequently, invoking, say, six methods with no parameters contributes the same to class coupling as invoking two methods each with two parameters or invoking one method with five parameters. This implies a special empirical model, i.e., weighting method invocations by the number of parameters passed, which should be investigated empirically.

4.2.4 Summary

Table 2 summarizes the discussion of measures. For each measure, we indicate the type of coupling it uses, what factors determine the strength of coupling, if it is an import or export coupling measure, if indirect coupling is accounted for, and how inheritance is dealt with (inheritance-based coupling, noninheritance-based coupling, or both). The columns C2, C4, and C5 show violations of the coupling properties Coupling.2, Coupling.4 and Coupling.5, where an "X" indicates a violation (properties Coupling.1 and Coupling.3 are fulfilled by all measures and are therefore not listed in the table).

5 A Unified Framework for Coupling Measurement

In this section, a new framework for coupling in object-oriented systems is proposed. The framework is defined on the basis of the issues identified by comparing existing coupling frameworks (Section 4.1.2) and the discussion of existing measures with respect to these issues (Section 4.2.2). The objective of the unified framework is to support the comparison and selection of existing coupling 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 coupling for which there are few or no measures defined.

TABLE 2
Properties of Coupling Measures

Measure	Type of coupling	Strength of coupling	Import or export coupling	Indirect coupling	Inheritance	C2	C4	C5
CBO	method invocation, attribute reference	#coupled classes	both (indifferent)	no	both			x
CBO´					non-inh.-based			x
RFC_α	method invocation	#methods invoked	import	depends	both	x		x
RFC				no		x		x
RFC´				yes		x		x
MPC	method invocation	#method invocations	import	no	both			
DAC	type of attribute	#attributes	import	no	both	x		
DAC´		# distinct types	import	no		x		x
COF	method invocation, attribute reference	#coupled classes	both (distinguished)	no	non-inh.-based		x	x
ICP	method invocation	#method invocations, #parameters passed	import	no	both			
IH-ICP					inh.-based			
NIH-ICP					non-inh.-based			
IFCAIC	type of attribute	#attributes	import	no	non-inh.-based			
ACAIC			import	no	inh.-based			
OCAIC			import	no	non-inh.-based			
FCAEC			export	no	non-inh.-based			
DCAEC			export	no	inh.-based			
OCAEC			export	no	non-inh.-based			
IFCMIC	type of parameter	#of parameters	import	no	non-inh.-based			
ACMIC			import	no	inh.-based			
OCMIC			import	no	non-inh.-based			
FCMEC			export	no	non-inh.-based			
DCMEC			export	no	inh.-based			
OCMEC			export	no	non-inh.-based			
OMMIC	method invocation, passing of pointer to method	#method invocations, #pointers passed to a method	import	no	non-inh.-based			
IFMMIC			import	no	non-inh.-based			
AMMEC			import	no	inh.-based			
OMMEC			export	no	non-inh.-based			
FMMEC			export	no	non-inh.-based			
DMMEC			export	no	inh.-based			

The framework consists of six 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 coupling measures in Section 4.2. We then discuss how the framework can be used to derive coupling 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. Finally, we provide a set of guidelines how a given measurement goal influences this choice.

The six criteria of the framework are:

1) The *type of connection*, i.e., what constitutes coupling.
2) The *locus of impact*, i.e., import or export coupling.
3) *Granularity of the measure*: the domain of the measure and how to count coupling connections.
4) *Stability of server*.
5) *Direct or indirect coupling*.
6) *Inheritance*: inheritance-based vs. noninheritance-based coupling, and how to account for polymorphism, and how to assign attributes and methods to classes.

These criteria are necessary to consider when specifying a coupling measure. However, they are not sufficient, as other aspects such as properties of measures (e.g., those discussed in Section 4.2.3) 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

Choosing a type of connection implies choosing the mechanism that constitutes coupling between two classes. Table 3 summarizes the possible types of connections, i.e., links between a client and a server "item" (attribute, method, or class), and those used by the reviewed measures. The items are listed in the columns "client item" and "server item." Column "Description" explains the type of connection. Column "Design phase" indicates from which design phase on the type of connection typically is applicable. The numbers in column "#" are used later to reference the types of connections. Column "Measures" lists for each type of connection, which coupling measures use that type of connection.

5.1.2 Locus of Impact

It has to be decided whether to count import or export coupling:

- Import coupling analyzes attributes, methods, or classes in their role as clients (users) of other attributes, methods, or classes.
- Export coupling analyzes the attributes, methods, and classes in their role as servers to other attributes, methods, or classes.

Table 4 shows which coupling measures are import coupling measures and which are export coupling measures.

5.1.3 Granularity

The granularity of the measure is the level of detail at which information is gathered. The granularity of the measure is determined by two factors:

1) the domain of the measure, i.e., what components are to be measured
2) how exactly the connections are counted.

Domain of measure. Table 5 shows possible domains for the coupling measures and which measures from Section 4.2 have that domain. Of course, measures for smaller domains such as the class level can easily be extended to larger domains like the sets of classes and the system level.

How to count connections. The next decision is how to count connections. Available options for this decision can be restricted by the domain of the measure. For measures defined at the method or attribute level, two options are listed in Table 6. Column "#" provides a label used for reference purposes, and "Description" explains the option. Columns "Import coupling example" and "Export coupling example" illustrate the options using the example of references to attributes by methods (connection type 5).

The difference between options A) and B) is that multiple connections between two items are counted separately in option A), and counted as one in option B).

At the class level, there are four options to count connections, shown in Table 7.

The difference between options D) and E) is that if, for instance, two methods of a class reference the same attribute, the references are counted separately (once for each method) according to D), and counted as one for the class according to E).

In Table 7, we use phrases such as "the methods of the class" or "the attributes of the class." This is imprecise, because it is not yet specified if a method or attribute has to be declared or implemented in the class in order to belong to it. This issue will be discussed in criterion 6 (inheritance).

Measures defined for sets of classes or the system can be constructed by adding up the number of connections of the relevant classes, counted according to one of the options C) to F).

The coarseness of the resulting measures increases gradually from option A) to option F): option A) produces measures at the finest grain, option F) produces measures at the coarsest grain.

Table 8 shows how coupling measures count frequencies of connections.

5.1.4 Stability of Server

For the discussion of this criterion, two different categories of class stability are defined:

- *unstable classes*: These are classes which are subject to development or modification in the project at hand. Unstable classes are problem domain classes which are being developed exclusively for the system, or are being adapted from other systems (reuse with modification).
- *stable classes*: Classes that are not subject to change in the project at hand. Stable classes are classes imported from libraries, or classes reused verbatim from other systems.

TABLE 3
TYPES OF CONNECTION

#	Client item	Server Item	Description	Design phase	Measures
1	attribute *a* of a class *c*	class *d*, $d \neq c$	class *d* is the type of *a*	HLD	DAC, DAC´, IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC
2	method *m* of a class *c*	class *d*, $d \neq c$	class *d* is the type of a parameter of *m*, or the return type of *m*	HLD	IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, OCMEC
3	method *m* of a class *c*	class *d*, $d \neq c$	class *d* is the type of a local variable of *m*	LLD	-
4	method *m* of a class *c*	class *d*, $d \neq c$	class *d* is the type of a parameter of a method invoked by *m*	LLD	-
5	method *m* of a class *c*	attribute *a* of a class *d*, $d \neq c$	*m* references *a*	HLD	CBO, CBO´, COF
6	method *m* of a class *c*	method m' of a class *d*, $d \neq c$	*m* invokes m'	HLD	CBO, CBO´, RFC_α, RFC, RFC´, MPC, COF, ICP, NIH-ICP, IH-ICP, OMMIC, IFMMIC, AMMIC, OMMEC, FMMEC, DMMEC
7	class *c*	class *d*, $d \neq c$	high-level relationships between classes, such as “uses”, “consists-of”	An	-

TABLE 4
IMPORT AND EXPORT COUPLING MEASURES

Direction	Measures
Import	CBO, CBO´, RFC_α, RFC, RFC´, MPC, DAC, DAC´, COF, ICP, IH-ICP, NIH-ICP, IFCAIC, ACAIC, OCAIC, IFCMIC, ACMIC, OCMIC, IFMMIC, AMMIC, OMMIC
Export	CBO, CBO´, COF, FCAEC, DCAEC, OCAEC, FCMEC, DCMEC, OCMEC, OMMEC, FMMEC, DMMEC

TABLE 5
MAPPING OF MEASURES TO DOMAINS

Domain	Measures
attribute	-
method	ICP, IH-ICP, NIH-ICP
class	CBO, CBO´, RFC_α, RFC, RFC´, MPC, DAC, DAC´, ICP, IH-ICP, NIH-ICP, IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC, IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, OCMEC, OMMIC, IFMMIC, AMMIC, OMMEC, FMMEC, DMMEC
set of classes	ICP, IH-ICP, NIH-ICP
system	COF

TABLE 6
OPTIONS FOR COUNTING CONNECTIONS AT THE ATTRIBUTE AND METHOD LEVEL

#	Description	Import coupling example	Export coupling example
A)	count individual connections	for each method, the number of references to attributes	for each attribute the number of references to the attribute
B)	count the number of distinct items at the other end of the connections	for each method, the number of attributes referenced	for each attribute the number of methods that reference the attribute

TABLE 7
OPTIONS FOR COUNTING CONNECTIONS AT THE CLASS LEVEL

#	Description	Import coupling example	Export coupling example
C)	add up the number of connections counted as in A) for each method or attribute of the class	the total number of attribute references by methods in the class	the total number of references to attributes of the class
D)	add up the numbers of connections counted as in B) for each method or attribute of the class	add up the number of attributes referenced by each method of the class	add up or for each attribute of the class: the number of methods that reference the attribute
E)	count the number of distinct items at the end of connections starting from or ending in methods or attributes of the class	the number of attributes referenced by the methods of the class	the number of methods referencing attributes of the class
F)	for a class *c*, count the number of other classes to which there is at least one connection	the number of classes which have an attribute that is referenced by a method of class *c*	the number of classes which have a method that reference an attribute of class *c*

TABLE 8
MAPPING OF MEASURES TO OPTIONS
FOR COUNTING THE FREQUENCY OF CONNECTIONS

Option #	Measures
A)	ICP, IH-ICP, NIH-ICP
B)	-
C)	MPC, DAC, ICP, IH-ICP, NIH-ICP, IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, OCMEC, OMMIC, IFMMIC, AMMIC, OMMEC, FMMEC, DMMEC
D)	DAC´
E)	RFC_α, RFC, RFC´
F)	CBO, CBO´, COF

Using a class (the server class) which is unstable is different from using a stable server class. If an unstable server class is modified, this may require the using class to be modified as well. This modification in turn may trigger other modifications, and so on. Since a stable server class is not subject to modification, it cannot trigger an avalanche of changes that cascade through the system.

Other categorizations than the above are conceivable (e.g., verbatim reused classes, classes where few changes are expected, classes where many changes are expected, etc.). However, we suggest to use a categorization scheme where the decision, into which category a given class belongs, can be made automatically. Otherwise, the resulting coupling measures may no longer be automatically collectable.

Using the above categorization, we have basically four options to distinguish stability of the server class. These are summarized in Table 9.

Class stability has not been addressed in the definition of any of the reviewed measures. In other words, the measures make no distinction of the stability of the server class, i.e., option III.

5.1.5 Direct or Indirect Connections

We have to decide whether to count direct connections only or also indirect connections. For example, if a method m_1 invokes a method m_2, which in turn invokes a method m_3, we can say that m_1 indirectly invokes m_3. Methods m_1 and m_3 are indirectly connected.

RFC_α and RFC' are the only measures to take indirect connections into account. All other measures count direct connections only.

5.1.6 Inheritance

Three aspects need to be considered with respect to inheritance:

- Is there a need to distinguish between inheritance-based coupling and noninheritance based coupling?
- How do we assign methods and attributes to classes?
- For method invocations: shall we consider static or polymorphic invocations?

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

Inheritance-based vs. noninheritance-based coupling. First, we have to decide whether to count inheritance-based coupling and/or noninheritance-based coupling. Inheritance-based coupling analyzes connections between classes that are related via inheritance. Likewise, noninheritance-based coupling refers to connections between classes that are not related via inheritance. We have four options for dealing with inheritance, which are described in Table 10. In column "measures," we also list which measures in Section 4.2 conform to the respective option.

There are no measures for option III, because a single measure cannot count both inheritance-based and nonin-

TABLE 9
OPTIONS TO ACCOUNT FOR STABILITY OF SERVER CLASS

Option #	Description
I	Take only connections from or to unstable classes into account, do not count connections from or to stable classes.
II	Take only connections from or to stable classes into account.
III	Take all connections into account, regardless of the stability of the class at the other end, i.e., do not distinguish stability at all.
IV	Count separately connections to stable and unstable classes. This results in two sets of measures. For example, on the class level, we would have two coupling values for each class.

TABLE 10
OPTIONS FOR INHERITANCE-BASED COUPLING

Option #	Description	Measures
I	count inheritance-based coupling only	IH-ICP, ACAIC, DCAEC, ACMIC, DCMEC, AMMEC, DMMEC
II	count non-inheritance-based coupling only	CBO´, COF, NIH-ICP, IFCAIC, OCAIC, FCAEC, OCAEC, IFCMIC, OCMIC, FCMEC, OCMEC, IFMMIC, OMMEC, FMMEC
III	count inheritance-based and non-inheritance-based coupling separately	-
IV	count inheritance-based and non-inheritance-based coupling, making no distinction	CBO, RFC_{α}, RFC, RFC´, MPC, DAC, DAC´, ICP

heritance-based coupling separately (option III). To implement this option, pairs of measures are needed, where one measure of each pair conforms to option I, the other to option II (for instance, IH-ICP and NIH-ICP). (see Table 10.)

Polymorphism. The next question is how to deal with polymorphism. This is relevant for method invocations only (connections of type 6). There are two options:

1) Account for polymorphism, i.e., for a method m, we count connections between m and all methods $m' \in PIM(m)$.
2) Do not account for polymorphism, i.e., for a method m, we count connections between m and methods $m' \in SIM(m)$ only.

Table 11 shows which measures in Section 4.2 account for polymorphism and which do not. Only measures that are concerned with method invocations are considered.

How to assign methods and attributes to classes. The final question is to decide to which class an attribute or method belongs. We have to decide if inherited methods and attributes belong to the inheriting class or not. We distinguish between two cases:

- When we compute the coupling of a class, we have to determine what are the methods/attributes of the class, and therefore contribute to the coupling of the class. The available options are:
  - only methods and attributes implemented in the class contribute to the coupling of the class
  - all methods and attributes implemented or declared in the class contribute to the coupling of the class
- When we count the frequency of connections according to option F) (i.e., for a given class, we count the number of other classes it is connected to, cf. criterion 3), we have to assign the items at the other ends of the connections to a class. The available options also depend on whether we are counting import or export coupling and are summarized in Table 12.

For the measures defined in Section 4.2, only methods and attributes implemented in a class contribute to the coupling of the class.

5.2 Application of the Framework

We apply the framework to select existing measures or to derive new measures for a given measurement goal. Application is performed by the 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. The criteria must be dealt with in the order introduced in Section 5.1 because, as explained below, a decision made for one criterion can restrict the available options for subsequent criteria.
- Choose the existing measures accordingly or, if none exist to match the decisions made, construct new coupling measures. Remember that properties such as those presented in Section 4.2.3 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 development phase at which measurement is to take place,
- the underlying hypothesis which drives measurement. The hypothesis will be of the form “Internal attribute coupling (as measured by the coupling measures defined) has a causal effect on external quality

TABLE 11
MAPPING OF MEASURES TO OPTIONS FOR ACCOUNTING FOR POLYMORPHISM

Type	Measure
account for polymorphism	CBO, CBO´, RFC_α, RFC, RFC´, COF, ICP, NIH-ICP, IH-ICP
do not account for polymorphism	MPC, OMMIC, IFMMIC, AMMIC, OMMEC, FMMEC, DMMEC

TABLE 12
OPTIONS TO ASSIGN METHODS AND ATTRIBUTES TO A CLASS

Import Coupling	Export Coupling
Assign server item to the class(es) where it is implemented (may be several classes if we account for polymorphism).	Assign client item to the class where it is implemented.
Assign server item to the class used to reference it.	---

attribute Y." The external attribute Y could be maintainability, reliability, etc. As discussed in [10], 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 system 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 coupling measures. Having an operational definition of the external quality attribute may help in the processes of choosing the appropriate coupling measures.

We now discuss, for each criterion, how the external attribute and the target development phase from a given measurement goal influences the choice of the available options and, where applicable, other aspects that may also impact this choice. We then illustrate the process of constructing an appropriate coupling measure by means of an example.

5.2.1 Type of Connection

Influence of development phase. The selection of one or more types clearly will be influenced by the development phase at which the measure is aimed. We are confined to types of connections that are applicable at the target development phase. In Table 3, we indicated for each type of connection from which development phase on it is applicable.

Influence of external attribute. It is difficult to provide guidelines for how a given external attribute affects the choice for one or more of the available types of connections. There are no obvious guidelines we could provide, and as of yet, only little practical experience has been gathered that we could report. Perhaps the exact definition of the external attribute gives some clue as to which types of connections will be relevant. To begin with, we recommend choosing several types of connections and conducting statistical analyzes to investigate which types of connections are relevant, i.e., support empirically the underlying hypothesis of the measurement goal.

Additional remarks. Different types of connections have different strengths. For instance, according to information hiding principles, referencing an attribute of another class is worse (i.e., stronger) than invoking a method of another class. Therefore, we recommend not to incorporate more than one type of connection into a single measure. Rather, separate measures for each type of connection should be used. Exceptions to this rule may be justified, if the underlying hypothesis of the measurement goal does not require that different types of connections be differentiated, or if the measure is coarse (cf. criterion 3, granularity). We will come back to this issue in the description of criterion 3.

5.2.2 Locus of Impact

Influence of development phase. None.

Influence of external attribute. High import coupling of, e.g., a class indicates that the class depends strongly on other classes and their methods and attributes. Import coupling may therefore be relevant in conjunction with the following external attributes:

- *Understandability*: To understand a method or class, we must know about the services the class uses.
- *Error-proneness*: For similar reasons as understandability: if we incorrectly use an external service because we misunderstand it, we are likely to introduce errors.
- *Maintainability*: Low understandability and high error-proneness result in low maintainability.
- *Reusability*: If a class depends on a large amount of external services, it will be more difficult to reuse it in other systems (because the external services will have to be made available too in the other systems).

High export coupling of, e.g., a class means that the class is used a lot by other classes and their methods and attributes. Export coupling may be relevant in conjunction with the following external attributes:

- *Criticality*: Any defects in a class with high export coupling are more likely to propagate to other parts of the system. Such defects are more difficult to isolate. In that respect, classes with high export coupling are particularly critical. An export coupling measure

could therefore be used to select classes that should undergo special (effective but costly) verification or validation processes.

- *Testability*: A class with high export coupling can be difficult to test. If defects need to propagate to other parts of the system to cause failures there, they may not be detected when testing the class in isolation.

5.2.3 Granularity

Domain of the measure. The finest possible domain of the measure has already been determined by criteria 1 and 2, and is evident from Table 3: It is either the client or server item of the chosen type of connection. For example, if we decide to measure import coupling and connections of type 5 (references from methods to attributes), the finest possible domain of the measure is the method (see row 5, column "client item"): for each method, count how often attributes are referenced by the method. Likewise, if we decide to measure export coupling and connections of type 1 (aggregation), the finest possible domain of the measure is the class: for each class, how often is it used in other classes as type of an attribute.

Influence of development phase. The development phase influences the choice of domain indirectly only, in that it determines the finest possible domain of the measure. As explained above, the finest possible domain is determined by criteria 1 and 2. The target development phase is the main decisive factor for criterion 1. Thus, the target development phase has an indirect influence on the domain of the measure.

Influence of external attribute. Similar to the development phase, the external attribute has an indirect influence on the choice of domain. The finest possible domain is in part determined by criterion 2, for which the external attribute is the main decisive factor.

However, for some measurement goals we may want to choose a coarser domain for the measure than the finest possible. For instance, if the measurement goal is to characterize understandability, reusability etc. of a class, a measure defined at the class level is appropriate. This can be accomplished by taking into account all connections starting from or ending in the methods or attributes of the class. Similarly, measures defined on sets of classes or the whole system can be constructed by taking into account the connections starting from or ending in methods or attributes of the relevant classes. The ICP family of measures in Section 4.2 provides an example of how a measure defined on the method level (ICP(m)) is scaled up to the class level (ICP(c)) and set-of-classes level (ICP(SS)).

How to count connections. For this criterion, we have six options A) to F), where A) yields the finest measures and F) yields the coarsest measures; see Tables 6 and 7.

Influence of development phase. The less precise and stable the information about the connections, the coarser the measure should be. If the information about the actual connections is detailed enough at the design phase we aim at, we may count individual connections. If the information we have is less detailed, or likely to change in the future, we may want to use a coarser measure. Generally speaking, the later in the design process the application of the measure, the more precise and stable the available information, the finer the measure.

Based on commonly used design methods, our recommendation is to use options D) to F) for analysis and high-level design, and options A) to D) for low-level design and implementation.

Influence of external attribute. In some cases, we may only need coarser grain measures. For example, assume we want to characterize understandability of a class for which the source code is available. The hypothesis is that the more methods invoked from other classes, the lower the understandability. To test this hypothesis, we count the methods used by the class according to option E), even though the source code is available: According to the hypothesis, the number of methods is decisive for understandability, not how often the methods are used.

Additional remarks. The coarser the measure, the less it says about the actual strength of connections between classes. Therefore, a single coarse measure might just as well take into account several types of connections at once: even if different types of connections have different strengths, mixing the types is acceptable for coarse measures (options E) and F)), because the strength of connections is not accounted for by the measure. In Section 4.2, CBO and COF are examples for coarse measures (option F)) which take more than one type of connection into account.

As was demonstrated in Section 4.2.3, measures which count multiple connections between the same items as one do not fulfill the additive property Coupling.5. Options B), D), E), and F) count the frequency of connections in this manner. Therefore, we must not use options B), D), E), or F) if we want our measures to fulfill the additive property.

5.2.4 Stability of Server

Influence of Development Phase. None.

Influence of external attribute. In the following, we list for each option some examples illustrating when it may be appropriate.

- *Option I* (count connections to unstable classes only) could be used for measuring reusability. Typically, library classes are easy to reuse. Import coupling from library classes may therefore be neglected.
- *Option II* (count connections to stable classes only) could be used for change impact analysis, where a libraries are to be replaced by other, possibly more efficient, libraries. Import coupling from the old libraries will be an indicator for the effort required to update a system to the new libraries.
- *Option III* (no distinction of stable and unstable classes) could be used when analysing understandability. *Hypothesis*: understandability is influenced by the number of services used. It should not matter if the server classes are stable or not. (Additional assumption needed here: stable and unstable classes are equally well known).

- *Option IV* (count separately connections to both stable and unstable classes) could be used when analyzing maintainability. *Hypothesis*: Maintainability is influenced by dependencies on both stable and unstable classes, and dependencies on unstable classes weigh heavier. To verify this hypothesis, coupling with stable and unstable classes has to be measured separately.

5.2.5 Direct or Indirect Connections

Influence of Development Phase. None.

Influence of external attribute. Indirect connections can be relevant when estimating the effort for run-time activities such as testing and debugging, or to estimate the impact of a modification to a class *c* on the system: the modification may necessitate other modifications to classes directly and indirectly connected to class *c* (ripple effects).

Indirect connections may also be relevant for reusability: if a class *c* is to be reused in another system, not only the classes to which *c* is coupled have to be provided in the system, but also classes required by these coupled classes, etc.

Direct connections are sufficient for the analysis of understandability: To understand a class, we need to know the functionality of the services directly used by the class. We do not need to know how these services are implemented, and therefore, what other services these services need (at least, this is true if the services are well documented).

5.2.6 Inheritance

There are three options for inheritance that must be considered.

Inheritance-based vs. noninheritance-based coupling

Influence of development phase. None.

Influence of external attribute. It is difficult to provide guidelines here. There are no obvious guidelines, and as of yet, only little practical experience concerning the relative importance of inheritance-based and noninheritance-based coupling is available: Briand et al. [4] showed that inheritance-based coupling is not a significant predictor for fault-prone classes, whereas noninheritance-based coupling is.

To begin with, we recommend to measure both inheritance-based and noninheritance-based coupling separately (option III), and conduct statistical analyses to investigate their relative importance.

Polymorphism

Influence of development phase. None.

Influence of external attribute. To analyze understandability, we do not need to account for polymorphism: the methods that are invoked through a method invocation have the same signature and should provide the same functionality (if we know one method, we know them all).

Accounting for polymorphism could be important for the analysis of error-proneness and testability. Because each method in $PIM(m)$ has its own implementation, defects can propagate to or originate from any of these methods.

How to assign methods and attributes to classes

Influence of development phase. None.

Influence of external attribute. None.

Additional remarks. The choice for one of the available options is largely influenced by the decision made for the inheritance-based vs. noninheritance-based coupling criterion: if we count inheritance-based coupling (options I, III, or IV), "the attributes and methods of a class" have to be those implemented in the class. Because with inheritance-based coupling we measure the degree to which a class *c* is coupled to its ancestors, it makes no sense to assign the methods and attributes *c* has inherited to class *c*.

Assigning inherited methods and attributes to a class only makes sense when we analyze the coupling of a class *c* "as a whole" to other classes not related to *c* via inheritance. Connections from or to methods and attributes that class *c* inherits contribute to the coupling of class *c*. We cannot provide an example that shows the application of this option. However, this does not imply that there are no useful applications of this option. Therefore, we leave it as a part of the framework.

5.2.7 Construction of the Coupling Measures

After selection of one or more options for each criterion of the framework, we can construct a set of coupling measures accordingly. Let us assume that our measurement goal is to analyze the source-code of some system to predict maintenance effort where maintenance effort might be defined as the number of person hours spent fixing faults in a class or implementing changes to a class as a result of requirements changes. This data is measured over a certain period of time, say, one year. For each criterion of the framework, the following decisions are made.

- *Type of connection (criterion 1)*: method invocations (option 6).

  Justification: At the source code level, assuming the system is a pure object-oriented implementation, method invocations are hypothesized to be the most relevant type of connection.

- *Locus of impact (criterion 2)*: Count import coupling.

  Justification: If a method invokes many other methods, it is more likely to be affected by changes to the invoked methods.

- *Granularity (criterion 3)*:

  a) Required domain is "class."

     Justification: Maintenance effort is collected at the class level.

  b) Count individual connections (option C).

     Justification: The more often a method is invoked, the more effort is likely to be required to modify the invoking method when modification of the invoked method takes place.

- *Stability of server (criterion 4)*: only count connections to unstable classes (option I).

  Justification: Assume that stable classes are reliable and will not need modification even as a result of requirements changes.

- *Indirect or direct connections (criterion 5)*: Count both types of connections. Justification: there is no clear rationale for choosing one particular type of connection. All the available options should be investigated. For the sake of illustration, only direct connections are considered in Fig. 4.
- Inheritance (criterion 6):

  a) Count both inheritance-based and noninheritance-based coupling and distinguish between these types of coupling (option IV).

  Justification: We do not know whether inheritance-based or noninheritance-based coupling is more important. Therefore, we need to measure both types of coupling separately to investigate their relative importance.

  b) Account for polymorphism.

  Justification: Any method that is used by a class may give rise to a modification to the class. Therefore, we must include all methods that can be possibly invoked through polymorphism and dynamic binding.

  c) Only methods implemented in a class contribute to the coupling of the class.

  Justification: We must choose this option because we count inheritance-based coupling (criterion 6a above).

The corresponding coupling measures are shown in Fig. 4. For measure "ibc" (inheritance-based coupling), we indicate for each criterion where it is reflected in the definition of the measure. The only difference with measure "nibc" (noninheritance-based coupling) is caused by criterion 6a) and is also indicated in Fig. 4. For the distinction between stable and unstable classes, let us assume that we have a partition of the set C of all classes:

$$C = SC \cup UC,$$

where $SC \subseteq C$ is the set of stable classes in C, $UC \subseteq C$ is the set of unstable classes in C, and $SC \cap UC = \phi$.

5.2.8 Summary

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

- The measures generated with this framework are counts of connections between classes. This leads to the to the highest level of measurement, ratio measurement, 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 measurement of internal product attributes are not meaningful in isolation, but only if they capture relevant external quality attributes; for full details, see [10].
- Existing measures have been classified according to the options available for each criterion of the framework. This classification allows existing measures to be compared and their potential use identified. The classification has shown that some particular options of the framework criteria have no or only few corresponding measures proposed.
- Most object-oriented design methods define their own, specific types of connections between classes (e.g., "stimuli" in Jacobson's OOSE method [25] and "links" in Rumbaugh's OMT method [32]). The presented framework may be tailored to a particular design method by adding the types of connections that are unique to the design method to the list of possible types of connections (criterion 1).

6 Conclusions

Based on a standardized terminology and formalism, we have provided a framework for the comparison, evaluation, and definition of coupling measures in object-oriented systems. This framework is intended to be exhaustive and integrates new ideas with existing measurement frameworks

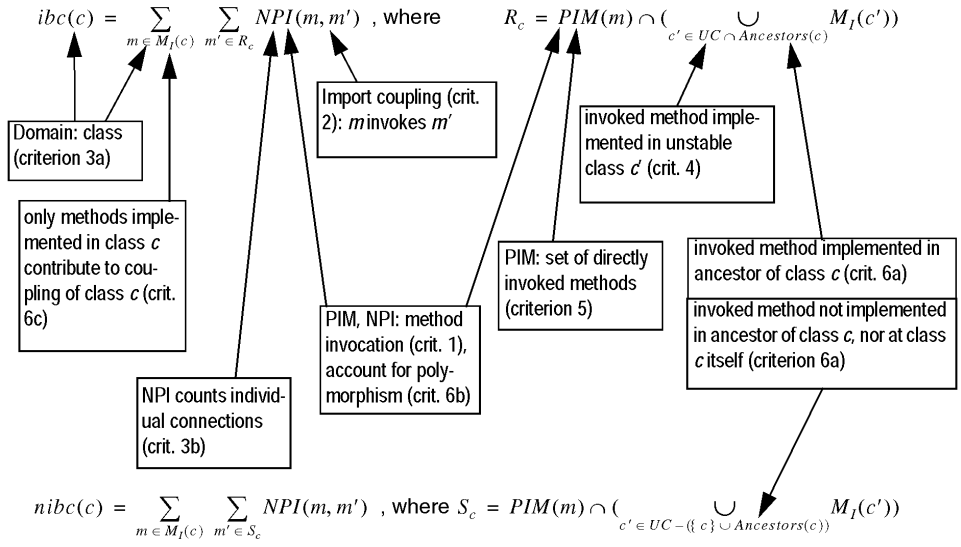


Fig. 4. Example of how the criteria are reflected in a measures' definitions.

in the literature. Thus, detailed guidance is provided so that coupling 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 coupling 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.

In the future, we will explore the measurement of dynamic aspects for coupling between objects at run-time, especially in the context of early design documents. Experienced developers think in terms of objects when they design. Therefore, the dynamic object structure at run-time may affect the cognitive complexity of the design.

As a complement to this paper and using systems developed by students, we performed a comprehensive empirical validation of the coupling measures discussed in this paper, as well as various cohesion and inheritance measures [5], [6], [7]. We plan to replicate this study on systems developed in industrial environments [37].

APPENDIX A—GLOSSARY OF TERMS

The terminology used for object-oriented concepts is provided as follows. Where applicable, the terminology defined by Churcher and Shepperd [17] has been used.

- *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*.
- *Aggregation*: "Part-Whole" relationship between two classes. A class *c* has an attribute of type class *d*. That attribute of type class *d* is a part, an aggregate, of class *c*. Other names: "containing," "has-a," or "consists-of" relationship, composition.
- *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. Other names: "pure virtual" in C++ lingo.
- *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).

We also define the terms client and server class, that are useful when discussing coupling.

- *Client class, server class*: In the context of coupling, it is convenient to distinguish the class that is using another class, and the class that is being used. We refer to the using class as client class, or client, and to the used class as server class, or server.

  Note that the means by which the server class is being used may be any of the types of connections identified in Section 5.1.1. Also, the roles of server and client must always be seen relative to a concrete connection between classes. For instance, if a class *c* invokes a method of some class *d*, class *c* is the client in this particular method invocation, and *d* is the server. If class *d* also invokes a method of class *c*, then class *d* is the client for that method invocation, and *c* is the server.

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 [10]:

- *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 $\mu: D \rightarrow R$, 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.
- *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.

APPENDIX B—A GENERIC OBJECT-ORIENTED DEVELOPMENT PROCESS

In this paper, we classify the coupling measures and the various types of coupling connections according to the phase, during the development process, where they become known or applicable. To be able to do a consistent classification, 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 [25]. 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.

APPENDIX C—OVERVIEW OF COUPLING MEASURES

Table 13 provides an overview of the coupling measures discussed in this paper. The meaning of the columns is as follows:

- *Name*: The name of the measure.
- *Definition*. The definition of the measure we derived in Section 4.2.1 using the defined formalism.
- *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 formal definition of the measure in Section 4.2.1.
- *Objectivity (subjective or objective)*: For an objective measure, the collected measurement data does not depend on the person collecting the data, i.e., the measure is automatable. For a subjective measure, the measurement data depends on the person collecting it and hence the measure 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 [21]. However, the empirical relation system used for the attribute is rarely provided. If it is not provided, the indicated scale type reflects our intuitive judgment.
- *Available at* (An / HLD / LLD / Imp): This column and column "Stable at" 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 B.

  A measure is *available* 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 available. Measurement values obtained at a development phase where the measure is available are only approximations; their values are likely to change in subsequent development phases.
- *Stable at* (An / HLD / LLD / Imp): A measure is stable 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 stable.
- *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 attribute of a class, subsystem, or system. Note that the theoretical validation of the measures in Section 4.2.3, and the recent empirical validation of all measures in [5], [6], [7] are not considered in this column. Thus, the column reflects the state-of-the-art before we performed this research.
  - *Source*: Literature references where the measure has been proposed.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thorough reviews and helpful comments, and the associate editor Dr. Dewayne Perry for his guidance and advice. This research has been conducted within the framework of Jürgen Wüst's masters thesis on quality measures for object-oriented systems. This research was supported, in part, by Daimler Benz Research Center, Ulm, Germany. John Daly was with Fraunhofer IESE when this research was performed.

TABLE 13
COUPLING MEASURES

Name	Definition	Operational definition	Objectivity	Level of measurement	Available at	Stable at	Language-specific	Validation	Source
CBO (coupling between object classes)	$CBO(c) = \lvert\{ d \in C - \{c\} \mid uses(c,d) \vee uses(d,c)\}\rvert$	no	obj	ratio	An	Imp	no	th & emp	[15]
CBO´	$CBO'(c) = \lvert\{ d \in C - (\{c\} \cup Ancestors(C)) \mid uses(c,d) \vee uses(d,c)\}\rvert$	no	obj	ratio	An	Imp	no	th & emp	[14]
RFC_α (response for class)	$RFC_\alpha(c) = \left\lvert \bigcup_{i=0}^{\alpha} R_i(c) \right\rvert$, for $\alpha = 1, 2, 3, \ldots$, where $R_0(c) = M(c)$ and $R_{i+1}(c) = \bigcup_{m \in R_i(c)} PIM(m)$	no	obj	ord	HLD	Imp	no	no	[17]
RFC	$RFC(c) = RFC_1(c)$	no	obj	ord	HLD	Imp	no	th & emp	[15]
RFC´	$RFC'(c) = RFC_\infty(c)$	no	obj	ord	HLD	Imp	no	th	[14]
MPC (message passing coupling)	$MPC(c) = \sum_{m \in M_I(c)} \sum_{m' \in SIM(m) - M_I(c)} NSI(m,m')$	no	obj	ratio	LLD	Imp	no	emp	[26]
DAC (data abstraction coupling)	$DAC(c) = \lvert\{ a \mid a \in A_I(c) \wedge T(a) \in C\}\rvert$	no	obj	ratio	An	LLD	no	emp	[26]
DAC´	$DAC'(c) = \lvert\{ T(a) \mid a \in A_I(c) \wedge T(a) \in C\}\rvert$	no	obj	ratio	An	LLD	no	emp	[26]
COF (coupling factor)	$COF(C) = \dfrac{\sum_{c \in C} \lvert\{ d \mid d \in C - (\{c\} \cup Ancestors(c)) \wedge uses(c,d)\}\rvert}{\lvert C\rvert^2 - \lvert C\rvert - \left(2 \sum_{c \in C} \lvert Descendents(c)\rvert\right)}$	no	obj	ord	An	Imp	no	no	[1]
ICP (information-flow-based coupling)	$ICP^c(m) = \sum_{m' \in PIM(m) \quad (M_{NEW}(c) \cup M_{OVR}(c))} (1 + \lvert Par(m')\rvert) \cdot NPI(m,m')$, $ICP(c) = \sum_{m \in M_I(c)} ICP^c(m)$, and $ICP(SS) = \sum_{c \in SS} ICP(c)$	yes	obj	ratio	LLD	Imp	no	th	[28]

TABLE 13
COUPLING MEASURES (CONTINUED)

Name	Definition	Operational definition	Objectivity	Level of measurement	Available at	Stable at	Language-specific	Validation	Source
NIH-ICP (information-flow-based non-inheritance coupling)	$NIH\text{-}ICP^c(m) = \sum_{m' \in R} (1 + \lvert Par(m')\rvert) \cdot NPI(m,m')$, where $R = PIM(m) \cap \left(\bigcup_{c' \in Ancestors(c)} M(c')\right)$, $NIH\text{-}ICP(c) = \sum_{m \in M_I(c)} NIH\text{-}ICP^c(m)$, and $NIH\text{-}ICP(SS) = \sum_{c \in SS} NIH\text{-}ICP(c)$	yes	obj	ratio	LLD	Imp	no	th	[28]
IH-ICP (information-flow-based inheritance coupling)	$IH\text{-}ICP^c(m) = \sum_{m' \in R} (1 + \lvert Par(m')\rvert) \cdot NPI(m,m')$, where $R = PIM(m) \cap \left(\bigcup_{c' \in C \quad (\{c\} \cup Ancestors(c))} M(c')\right)$, $IH\text{-}ICP(c) = \sum_{m \in M_I(c)} IH\text{-}ICP^c(m)$, and $IH\text{-}ICP(SS) = \sum_{c \in SS} IH\text{-}ICP(c)$	yes	obj	ratio	LLD	Imp	no	th	[28]
IFCAIC	$IFCAIC(c) = \sum_{d \in Friends^{-1}(c)} CA(c,d)$, where $CA(c,d) = \lvert\{ a \mid a \in A_I(c') \wedge T(a) = d\}\rvert$	yes	obj	ratio	An	LLD	C++	th & emp	[4]
ACAIC	$ACAIC(c) = \sum_{d \in Ancestors(c)} CA(c,d)$	yes	obj	ratio	An	LLD	no	th & emp	[4]
OCAIC	$OCAIC(c) = \sum_{d \in Others(c) \cup Friends(c)} CA(c,d)$, where $Others(c) = C - (Ancestors(c) \cup Descendents(c) \cup Friends(c) \cup Friends^{-1}(c) \cup \{c\})$	yes	obj	ratio	An	LLD	C++	th & emp	[4]
FCAEC	$FCAEC(c) = \sum_{d \in Friends(c)} CA(d,c)$	yes	obj	ratio	An	LLD	C++	th & emp	[4]
DCAEC	$DCAEC(c) = \sum_{d \in Descendents(c)} CA(d,c)$	yes	obj	ratio	An	LLD	no	th & emp	[4]
OCAEC	$OCAEC(c) = \sum_{d \in Others(c) \cup Friends^{-1}(c)} CA(d,c)$	yes	obj	ratio	An	LLD	C++	th & emp	[4]

TABLE 13
COUPLING MEASURES (CONTINUED)

Name	Definition	Operational definition	Objectivity	Level of measurement	Available at	Stable at	Language-specific	Validation	Source
IFCMIC	$IFCMIC(c) = \sum_{d \in Friends^{-1}(c)} CM(c,d)$, where $CM(c,d) = \sum_{m \in M_{NEW}(c)} \lvert\{a \mid a \in Par(m) \wedge T(a) = d\}\rvert$	yes	obj	ratio	HLD	LLD	C++	th & emp	[4]
ACMIC	$ACMIC(c) = \sum_{d \in Ancestors(c)} CM(c,d)$	yes	obj	ratio	HLD	LLD	no	th & emp	[4]
OCMIC	$OCMIC(c) = \sum_{d \in Others(c) \cup Friends(c)} CM(c,d)$	yes	obj	ratio	HLD	LLD	C++	th & emp	[4]
FCMEC	$FCMEC(c) = \sum_{d \in Friends(c)} CM(d,c)$	yes	obj	ratio	HLD	LLD	C++	th & emp	[4]
DCMEC	$DCMEC(c) = \sum_{d \in Descendents(c)} CM(d,c)$	yes	obj	ratio	HLD	LLD	no	th & emp	[4]
OCMEC	$OCMEC(c) = \sum_{d \in Others(c) \cup Friends^{-1}(c)} CM(d,c)$	yes	obj	ratio	HLD	LLD	C++	th & emp	[4]
IFMMIC	$IFMMIC(c) = \sum_{d \in Friends^{-1}(c)} MM(c,d)$, where $MM(c,d) = \sum_{m \in M_I(c)} \sum_{m' \in M_{NEW}(d) \cup M_{OVR}(d)} (NSI(m,m') + PP(m,m'))$	yes	obj	ratio	LLD	Imp	C++	th & emp	[4]
AMMIC	$AMMIC(c) = \sum_{d \in Ancestors(c)} MM(c,d)$	yes	obj	ratio	LLD	Imp	C++	th & emp	[4]
OMMIC	$OMMIC(c) = \sum_{d \in Others(c) \cup Friends(c)} MM(c,d)$	yes	obj	ratio	LLD	Imp	no	th & emp	[4]
FMMEC	$FMMEC(c) = \sum_{d \in Friends(c)} MM(d,c)$	yes	obj	ratio	LLD	Imp	C++	th & emp	[4]
DMMEC	$DMMEC(c) = \sum_{d \in Descendents(c)} MM(d,c)$	yes	obj	ratio	LLD	Imp	C++	th & emp	[4]
OMMEC	$OMMEC(c) = \sum_{d \in Others(c) \cup Friends^{-1}(c)} MM(d,c)$	yes	obj	ratio	LLD	Imp	no	th & emp	[4]

REFERENCES

[1] F. Abreu, M. Goulão, and R. Esteves, "Toward the Design Quality Evaluation of Object-Oriented Software Systems," *Proc. Fifth Int'l Conf. Software Quality*, Austin, Texas, Oct. 1995.

[2] V. Basili, L. Briand, and W. Melo, "Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems," *Comm. ACM*, vol. 39, no. 10, 1996.

[3] V. Basili, L. Briand, and W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Trans. Software Eng.*, vol. 22, no. 10, pp. 751-761, 1996.

[4] L. Briand, P. Devanbu, and W. Melo, "An Investigation into Coupling Measures for C++," *Proc. 19th Int'l Conf. Software Eng., ICSE '97*, Boston, pp. 412-421, May 1997.

[5] L. Briand, J. Daly, V. Porter, and J. Wüst, "A Comprehensive Empirical Validation of Product Measures in Object-Oriented Systems," Technical Report ISERN-98-07, Fraunhofer Inst. for Experimental Software Engineering, Germany, 1998. http://www.iese.fhg.de/ISERN/pub/isern_biblio_tech.html

[6] L. Briand, J. Daly, V. Porter, and J. Wüst, "Predicting Fault-Prone Classes with Design Measures in Object-Oriented Systems," *Proc. Ninth Int'l Symp. Software Reliability Eng., ISSRE'98*, Paderborn, Germany, Nov. 1998.

[7] L. Briand, J. Daly, V. Porter, and J. Wüst, "A Comprehensive Empirical Validation of Product Measures in Object-Oriented Systems," *Proc. Fifth Int'l Symp. Software Metrics, Metrics '98*, Bethesda, Md., Nov. 1998.

[8] L. Briand, J. Daly, and J. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.: An Int'l J.*, vol. 3, no. 1, pp. 65-117, 1998.

[9] L. Briand, J. Daly, and J. Wüst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," Technical Report ISERN-96-14, 1996.

[10] L. Briand, K. El Emam, and S. Morasca, "Theoretical and Empirical Validation of Software Product Measures," Technical Report ISERN-95-03, 1995. http://www.iese.fhg.de/isern/pub/isern_biblio_tech._html

[11] L. Briand, S. Morasca, and V. Basili, "Measuring and Assessing Maintainability at the End of High-Level Design," *Proc. IEEE Conf. Software Maintenance*, Montreal, Sept. 1993.

[12] L. Briand, S. Morasca, and V. Basili, "Defining and Validating High-Level Design Metrics," Technical Report CS-TR 3301, Univ. of Maryland, 1994; to be published in *IEEE Trans. Software Eng.*

[13] L. Briand, S. Morasca, and V. Basili, "Property-Based Software Engineering Measurement," *IEEE Trans. Software Eng.*, vol. 22, no. 1, pp. 68-86, 1996.

[14] S.R. Chidamber and C.F. Kemerer, "Towards a Metrics Suite for Object Oriented Design," A. Paepcke, ed., *Proc. Conf. Object-Oriented Programming: Systems, Languages and Applications, OOPSLA'91*, Oct. 1991. Also published in *SIGPLAN Notices*, vol. 26, no. 11, pp. 197-211, 1991.

[15] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, 1994.

[16] N.I. Churcher and M.J. Shepperd, "Comments on 'A Metrics Suite for Object-Oriented Design,'" *IEEE Trans. Software Eng.*, vol. 21, no. 3, pp. 263-265, 1995.

[17] N.I. Churcher and M.J. Shepperd, "Towards a Conceptual Framework for Object Oriented Software Metrics," *Software Eng. Notes*, vol. 20, no. 2, pp. 69-76, 1995.

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

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

[20] J. Eder, G. Kappel, and M. Schrefl, "Coupling and Cohesion in Object-Oriented Systems," Technical Report, Univ. of Klagenfurt, 1994. Also available at ftp://ftp.ifs.uni-linz.ac.at/pub/publications/1993/0293.ps.gz

[21] N.E. Fenton and S. Lawrence Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, 1996.

[22] M. Hitz and B. Montazeri, "Measuring Coupling and Cohesion in Object-Oriented Systems," *Proc. Int'l Symp. Applied Corporate Computing*, Monterrey, Mexico, Oct. 1995. A version of this paper (focusing on coupling only) has been published in *Object Currents*,

vol. 1, no. 4, *SIGS Publications*, 1996. http://www.sigs.com/publications/docs/oc

[23] M. Hitz and B. Montazeri, "Measuring Product Attributes of Object-Oriented Systems," W. Schöfer and P. Botella, eds., *Proc. ESEC '95 Fifth European Software Eng. Conf.*, Barcelona, Spain, Sept. 1995, Lecture Notes in Computer Science 989, Springer-Verlag, 1995.

[24] M. Hitz and B. Montazeri, "Chidamber & Kemerer's Metrics Suite: A Measurement Theory Perspective," *IEEE Trans. Software Eng.*, vol. 22, no. 4, pp. 276-270, 1996.

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

[26] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *J. Systems and Software*, vol. 23, no. 2, pp. 111-122, 1993.

[27] W. Li, S. Henry, D. Kafura, and R. Schulman, "Measuring Object-Oriented Design," *J. Object-Oriented Programming*, vol. 8, no. 4, pp. 48-55, 1995.

[28] Y.-S. Lee, B.-S. Liang, S.-F. Wu, and F.-J. Wang, "Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow," *Proc. Int'l Conf. Software Quality*, Maribor, Slovenia, 1995.

[29] R. Martin, "OO Design Quality Metrics–An Analysis of Dependencies," position paper, *Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94*, Oct. 1994.

[30] T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308-320, 1976.

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

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

[33] R.W. Selby and V.R. Basili, "Analyzing Error-Prone Systems Structure," *IEEE Trans. Software Eng.*, vol. 17, no. 2, pp. 141-152, 1991.

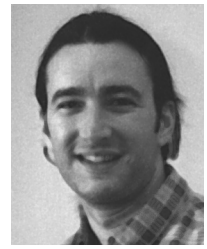
[34] R.C. Sharble and S.S. Cohen, "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods," *Software Eng. Notes*, vol. 18, no. 2, pp. 60-73, 1993.

[35] W. Stevens, G. Myers, and L. Constantine, "Structured Design," *IBM Systems J.*, vol. 13, no. 2, pp. 115-139, 1974.

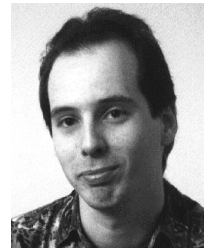
[36] P.A. Troy and S.H. Zweben, "Measuring the Quality of Structured Designs," *J. Systems and Software*, vol. 2, pp. 113-120, 1981.

[37] L. Briand, S. Ikonomovski, H. Lounis, and J. Wüst, "Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study," ISERN-28-29, *Proc. 21st Int'l Conf. Software Eng., ICSE'99*, 1999; to appear.

Lionel C. Briand received the BS degree in geophysics and the MS degree in computer science from the University of Paris VI, France. He received the PhD degree (with high honors) in computer science from the University of Paris XI, France. Briand is currently 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 object-oriented development techniques. He has published numerous articles in international conferences and journals and has been a program committee member or chair at several conferences such as ICSE, ICSM, ISSRE, METRICS, and SEKE. Before that, Dr. Briand 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 (Centre de Recherche Informatique de Montréal), Canada.



John W. Daly received the BSc and PhD degrees in computer science from the University of Strathclyde, Glasgow, Scotland, in 1992 and 1996, respectively. From 1996–1998, Daly was a software engineering researcher and then a research project manager in the Quality and Process Engineering Department at the Fraunhofer Institute for Experimental Software Engineering, Germany. In April 1998, he joined the Quality Assurance Department at Hewlett-Packard Ltd., South Queensferry, Scotland, as a software process engineer. His industrial activities and current research interests include software measurement, software process and improvement, software quality, and object-oriented development techniques.



Jürgen K. Wüst received the Diplom-Informatiker (MS) degree 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.