Measuring Coupling and Cohesion In Object-Oriented Systems

Martin Hitz, Behzad Montazeri

Institut für Angewandte Informatik und Systemanalyse, University of Vienna
hitz@ifs.univie.ac.at

Abstract - **As the role that software metrics in general and coupling in particular play with respect to maintainability of software products is widely accepted, current approaches to handle coupling and / or cohesion in object-oriented systems are evaluated. Having identified some inadequacies, we provide a comprehensive framework to deal with all sorts of coupling.**

This framework takes into account the distinction between object level - and class level coupling. This distinction refers to dynamic dependencies between objects on one hand and static dependencies between implementations on the other hand, representing important aspects of software quality at run-time and during the maintenance phase, respectively.

As far as cohesion is concerned, we analyze a well known metric put forward by Chidamber and Kemerer and re-stated by Li and Henry. As a result, we present a graph theoretic improved version of this metric.

Index Terms - **Object-oriented programming, software metrics, measure theory, coupling, cohesion, software maintenance**

I. Introduction

Trying to control software quality - and all related attributes as, e.g., reliability, maintainability, usability and so forth - it is obviously necessary to measure to what extend these attributes are achieved by a certain product. Such measurements are valuable both, in an *a posteriori* analysis of a finished product, and, even more important, in an *a priori* manner to guide the production process in order to avoid undesirable results in the first place. In this spirit, many software metrics have been established in the past, mainly in the area of traditional ("structured") software design. However, in the recent past a series of critiques of the methodological foundations of software measurement have been published (e.g., [9][10][25]), challenging some of the previously defined software metrics. Moreover, since the advent of the object-oriented paradigm in software engineering, even more problems have been identified with respect to applying traditional software metrics to object-oriented systems [4]. Thus, any attempt do define a measure for object-oriented software

- must take into account general measure theoretic foundations and
- must suit to the specific characteristics of object-oriented software.

With respect to the first requirement, we feel that in addition to the measure theoretic guidelines published, especially two important rules must be obeyed when attempting to measure internal[1] product attributes: Firstly, the attributes to be measured should have causal influences on some external attribute, and secondly, the corresponding measure must preserve all generally accepted empirical relations established.

In structured design and programming the importance of *coupling* and *cohesion* as main attributes related to the goodness of decomposition has been well understood; software engineering experts assure that designs with low coupling and high cohesion lead to products that are both, more reliable and more maintainable [10][16][21]. From the beginnings of this design paradigm [18], the various kinds of communication between modules have been discussed and ordered with respect to their effects on the quality of the design. This ordinal "metric" has obtained more or less general acceptance, even though it has taken a long time before a numeric measure had been proposed for this attribute [10]. In short, the following list introduces the different types of coupling in increasing order of malignity:

1. *Data Coupling* (communication via scalar parameters)
2. *Stamp Coupling* (dependency induced by the type of structured parameters)
3. *Control Coupling* (parameters are used to control the behavior of a module)
4. *Common Coupling* (communication via shared global data)
5. *Content Coupling* (one module shares and/or changes the definition of another nodule)

For object oriented software, the notion of coupling has not been considered with similar rigor by the pioneers who determined the major design guidelines of this paradigm. There are two main reasons for this negligence:

1. In structured design, there were few *semantic* guidelines to decompose a system into smaller subsystem. Consequently, syntactic aspects like size, coupling etc. played a major role. In contrast, in the object-oriented paradigm, the main criterion for systems decomposition is the mapping of objects of the problem domain into classes or subsystems in the analysis / design model, thus reducing the relative importance of syntactic criteria.
2. Object-oriented analysis and design strive to incorporate data and related functionality into objects. This strategy in itself certainly reduces coupling between objects. Therefore, explicitly controlling coupling does not seem to be as important as in structured (especially top-down) design.

[1]For a discussion of internal versus external attributes, see [10].

However, since employing object-oriented mechanisms in itself does not guarantee to really achieve minimum coupling, there is good reason to study coupling in object-oriented systems:

1. In many cases, data or operations cannot be unambiguously assigned to one or another class on the grounds of semantic aspects, thus designers do need some kind of additional criteria for such assignments.
2. Although introduction of classes as a powerful means for data abstraction reduces the data flow between abstraction units and therefore reduces also total coupling within a system, the number of variants of interdependency rises in comparison to conventional systems [24]. This can be attributed to
   - the variety of mechanisms (inheritance, delegation, using- and has-relationships etc.) and
   - the diversity of modules (classes and objects as well as functions and procedures in hybrid systems).

   The different mechanisms can sometimes also be employed interchangeably, e.g., inheritance can sometimes be simulated by delegation, or using-relationships can sometimes be replaced by has-relationships etc. Each of these variants exhibits different impacts on quality attributes which must be investigated and measured.
3. The principles of encapsulation and data abstraction, although fundamental to object-orientation, may be violated to different extents via the underlying programming language [7]. This leads to different strength of de-facto coupling which should be taken into account.

In this spirit, several researchers have tried to adopt the notion of coupling for the object-oriented paradigm, as will be discussed in Section A. Budd, for instance, demands that “Objects from distinct classes should have as little coupling as possible, not only to make them more understandable, but so that they may easily be extracted from a particular application and reused in new situations” [3]. Thus, coupling seems to be even more important in object-oriented systems:

- Coupling of client objects to a server object may introduce *change dependencies*. The tighter the coupling, the harder the effects on the clients whenever a crucial aspect of the server is being changed.
- High coupling between two objects makes it *harder to understand* one of them in isolation. In contrast, low coupling leads to self-contained and thus easy to understand, maintainable objects (“KISS”-principle: “keep it simple & stupid”).
- High coupling also increases the probability of *remote effects*, where errors in one object cause erroneous behavior of other objects. Again, lose coupling makes it easier to track down a certain error, which in turn improves testability and eases debugging.

In this paper, based on a general notion of coupling, we attempt to give appropriate definitions for coupling and cohesion in object-oriented systems and identify a collection of dimensions that should be taken into account upon measuring these attributes. Analyzing the effects of coupling, it turns out that these can naturally be partitioned into two classes attributed to two different variants of coupling, namely *object level coupling* and *class level coupling*, respectively.

Although our primary focus is on coupling as one of the most important internal attributes of software products, we must necessarily consider also *cohesion* because of the dual nature of these two attributes: Attempting to optimize a design with respect to coupling between abstractions (modules, classes, subsystems...) alone would trivially yield to a single giant abstraction with no coupling at the given level of abstraction. However, such an extreme solution can be avoided by considering also the antagonistic attribute cohesion (which would yield inadmissibly low values in the single-abstraction case).

In the remainder of this paper, after giving some definitions, we provide an overview of current approaches of how to define and handle coupling in object-oriented systems. Issues which are, in our opinion, not adequately dealt with in the literature are then being put into a comprehensive framework in Section B. Section C contains a preliminary attempt to define a coupling metric on an ordinal scale within the framework proposed, complemented by a discussion of some coupling related attributes in Section D. Finally, Section iv is devoted to measuring structural cohesion in object-oriented systems.

II. PRELIMINARIES

In this section we provide some prerequisites used throughout the rest of this paper. Definition 1 clarifies some object-oriented parlance, while the following definitions are supposed to give a preliminary idea of coupling in object oriented systems. These definitions will be refined in Section B.

Definition 1 (Object oriented concepts): We will use the terms *object* and *class* according to the usual object-oriented terminology: A class provides the definition of structure (*instance variables*) and behavior (*methods*) of similar kinds of entities, an object is an instance of its respective class. Classes may be organized in inheritance hierarchies as *super-* and *sub-*classes.

An object accessing another object will be called *client* while the object accessed will be referred to as *server*; the corresponding classes will be called *client class* and *server class*, respectively. This definition also applies when an object accesses parts inherited from its own superclass (the server class in this case).

By *access to the interface* of a class we refer to sending messages according to its method protocol only; in this way, instance variables can never be directly read or modified. On the other hand, the term *access to the implementation* describes the situation when an instance variable is immediately accessed, either via a direct reference (i.e., by name) or indirectly via a method returning a reference to that variable. ■

An ontological foundation of *coupling* has been given by Wand and Weber [22] as follows.

Definition 2: The *history* of a thing is defined as the set of ordered pairs <t, s> each recording the thing's state s at a given point t in time. Two things are *coupled* if and only if at least one of the things' history depends upon the other thing's history. ■

Starting from this, we strive for a notion of coupling in object-oriented systems suitable for the definition of a measure of *coupling strength*. Thus, all variants of coupling should be captured, as various kinds of coupling are likely to differ with respect to

- their contribution to the overall measure and to
- the phase in the development life cycle (i.e., design, implementation, maintenance) they are most relevant for.

Recalling that a history in the above sense results from a series of state changes, we identify two major categories of state and state change:

- The *state of an object* (in the usual sense) in an object-oriented application may change at run-time.
- The *state of an object's implementation* (i.e., its class specification and the program code of the corresponding methods) may change during the development life cycle.

Accordingly, in the following definitions we distinguish between object level coupling and class level coupling:

Definition 3: Object level coupling (OLC) represents the coupling (in the sense of Definition 2) resulting from state dependencies between objects during the run-time of a system. ■

Definition 4: Class level coupling (CLC) represents the coupling resulting from implementation dependencies in a system. ■

Although in most cases object level coupling implies class level coupling, it is important to distinguish both types and their respective coupling strengths. Establishing such measures will be our major task in what follows.

III. Coupling

A. Survey of previous work

Chidamber and Kemerer [4] give the first formal definition of coupling between classes. Transforming the definition by Wand and Weber [22], Chidamber and Kemerer conclude,

"... any evidence of a method of one object using methods or instance variables of another object constitutes coupling."

As a *metric* for coupling, they define CBO (Coupling Between Objects) as proportional to the number of non-inheritance related couples with other classes.

While this idea has been widely appreciated in the literature in principle (and has recently been republished [5]), some deficiencies have been identified, notably that it does not scale up to higher levels of modularization [17]. Moreover, we may note that

- coupling is defined as an attribute of pairs of objects, but as a metric, it is aggregated to the total number of couples that one class has with other classes, thus implicitly assuming that all basic couples are of equal strength. This does certainly not hold, e.g., when message passing is intermingled with direct access to foreign instance variables. This has generally been identified as the worst type of coupling, also in traditional design [10]. But even if only message passing is considered, several authors have distinguished various forms of coupling with different strengths [12][14]. For example, sending messages to objects is considered superior to sending messages to one of an object's components, even if the selection of such a component is accomplished by means of an access message.
- it is not clear whether messages sent to a part of *self* (i.e., an instance variable of class type) contribute to CBO or not. Strictly adhering to [22], messages to self (or parts thereof) do not alter a foreign object's history and therefore do not constitute coupling. However, it does lead to a certain amount of class level coupling, which cannot be attributed to coupling as defined by [4]. Li and Henry refer to this type of coupling as data abstraction coupling (DAC), measured by the number of instance variables having an abstract data type [13].
- by explicitly neglecting inheritance related connections, Chidamber and Kemerer exclude from their measure contributions attributed to immediate access to instance variables inherited from superclasses, a kind of coupling considered to be among the worst types of coupling [14][20][24].

Chidamber and Kemerer also define RFC (Response For a Class) as the union of the protocol a class offers to its clients and the protocols it requests from other classes. Measuring the total communication potential, this measure is obviously related to coupling and is not independent of CBO.

In [6], Coad and Yourdon dedicate three paragraphs to emphasize the importance of coupling in general and its influence to change dependencies within the system (without providing a metric). Interestingly, they distinguish between "connection between objects" and "connection between classes", but use the latter term to refer to generalization relationships between classes only.

Lieberherr et al. give a plausible practical guideline for "good" object-oriented design, partly also controlling some aspects of coupling ("Law of Demeter" [14][15]). However, being formulated as a "law" with a dichotomous character, it is not intended to be used as a measure of coupling strength. LeJacq elaborates on this guideline and proposes a corresponding ordinal metric [12].

Budd tries to reinterpret the classical concepts of coupling and cohesion in the context of object-oriented languages [3]. The terms *Internal Data Coupling* and *Global Data Coupling* seem to fit easily to the object-oriented paradigm, although we believe that "data coupling" is of minor importance in object-oriented systems where the main communication mechanism is message passing between objects rather than sharing data. A similar argument applies to *Control Sequence Coupling* which seems even more alien to object-oriented design. Of course, if such situations *do* arise, it is indeed a sign of bad object-oriented style. Interestingly, Budd does not at all discuss the role of *Stamp Coupling*, although it seems to be the one most relevant for object-oriented programming.

Budd's discussion indicates a general inadequacy we found in the literature where the traditional notion of coupling is being extended to object-orientation: Traditionally, operations are considered to be applied only to "data" in the classical sense but not to "objects". Data in the sense of structured programming usually belong to standard data types with a stable protocol where no change dependencies with respect to their implementation occur. In contrast, the usage of complex data types in object-oriented systems leads not only to coupling to data but also to *data types* (i.e., class level coupling).

B. A comprehensive framework for measuring object-oriented coupling

As a synthesis of the above discussion, we propose a framework into which any suitable measure for coupling in object-oriented systems should be embedded. The design of this framework is intended to avoid the deficiencies identified in the above section while incorporating most of the profitable results of previous research.

In contrast to approaches published so far, as already mentioned in Section ii, we emphasize the distinction between *coupling among objects* (CLO) and *coupling among classes* (CLC). CLC is especially important when considering maintenance or change dependencies within an application: Changes in a server class may call for corresponding changes in client classes. Reusability is obviously also affected by CLC. On the other hand, OLC is relevant for all kinds of run-time-oriented activities like testing and debugging.

In what follows, we will try to find a measure for CLC and OLC, respectively.

Class level coupling

In order to refine our preliminary definition of CLC (Definition 4), we need the following clarification of the notion of state of a class:

Definition 5: By *state of a class* we refer to the class definition and the program code of its methods, i.e., a version of the class implementation (cf. Section ii).[2] ■

Thus, changes in one class require changes in all classes coupled to it. This leads to

Definition 4': Class level coupling represents the coupling (in the sense of Definition 2) resulting from state dependencies between classes during the development life-cycle. ■

In the sequel, CC will denote the dependent client class and SC the server class being changed.

In order to assess the strength of coupling between classes CC and SC, we must measure the expected extent of required change in CC. For this purpose, we identify the following factors the strength of class level coupling is depending on:

- *Stability of SC*: If SC is considered stable, no changes are likely to occur and thus CC will not incur any dependent changes. Stable classes are normally imbedded in the environment of a programming language ("foundation classes"), but they could also be part of an established class library for a certain application domain. The latter case arises when the application concepts are well defined and the implementations of those concepts are stable.
  Consider, for instance, the classification of "attribute types" (i.e., types of instance variables) given by White which happens to be perfectly compatible with the notion of stability (in decreasing order) [23]:
  1. Basic types of the implementation language like integer or character.
  2. Small reusable classes like string, date, time etc.
  3. Problem domain classes.

  Messages to instance variables of the first and second group are likely to be disregarded with respect to coupling, while messages to instance variables of the third group are certainly considered by most subjective notions of coupling (cf. design guidelines stating that it might not even be necessary to depict relationships to group 2 classes in a design document!). Such an empirical relation system ("group 1 and 2 coupling is less important than group 3 coupling") can easily be explained by the different stability of the classes involved.
  In cases where SC is unstable, we have to consider two subcases:
  1. Only SC's *implementation* is subject to change without affecting its interface.
  2. SC's *interface* may be modified also.

  Obviously, case 2 is more harmful than the first, while totally stable classes represent the ideal case.
- As a second factor, we consider the *type of access to SC*: CC may either restrict its access to the interface of SC (as usually desired in object-oriented design) or may refer to at least one instance variable of SC. The breach of the important design rule of encapsulation certainly yields higher coupling values.
  The more assumptions about the server are made by a client, the tighter the coupling. CC can rely on the following informations regarding SC:
  I. Access to interface:
  1. Protocol (or type) of SC (CC must know the messages understood by SC)
  2. Class of SC. While usually only the type of the server is important, i.e., the actual server may be an instance of any class implementing this type, there are cases where clients explicitly refer to the concrete class of the server, thus effectively reducing potential polymorphism. As an example, in the context of languages that support value semantics (like C++), whenever an object is defined to be a value (rather than a reference), its concrete class must be known.

  II. Access to implementation:
  3. Protocol of the instance variables of SC. If the instance variable of SC itself is returned by a method of SC, only its protocol, but not its identification are necessary for the communication.
  4. Names of instance variables of SC. In this case, CC explicitly refers to a *specific* instance variable of SC.
- Lastly, the *scope of access to SC objects within CC* also influences the coupling strength: The extent of required

[2]In this definition, we do not deal with class variables and the "state of class" they define at run-time. Our notion of "state of class" is not defined at run-time.

change also depends on where SC objects are used inside CC: The larger the program area from which SC may potentially be referenced, the higher the expected volume of follow-up changes caused by changes to SC. The following variants are possible:

1. SC is the class of an instance variable of CC: References to SC may occur in any method of CC.
2. A local variable of type SC is used within a method of CC: Only this method must be checked when SC is changed.
3. SC is a superclass of CC: Similar to case 1.
4. SC is the type of parameter of a method of CC: Similar to case 2.
5. CC accesses a global variable of class SC: Again similar to the first case.

It is worth noting that some combinations of the above criteria are not of interest. For instance, in the case where SC is a totally stable class, we need not consider any other dimensions as far as CLC is concerned.

Object level coupling

Before dealing with object level coupling, we need the following auxiliary definition:

Definition 6: Consider two objects M and A. If and only if

a) A is a genuine aggregate of M, i.e., M is a subobject of A that can only exist as a part[3] of A, or

b) M is represented by a local variable of one of A's method, i.e., M can only exist during the activation period of that method, or

c) M is a subobject of A inherited from one of A's super classes,

then M is called a *native object of* A (or simply: M is *native to* A). ■

The state of native objects thus represent parts of the owner object's state. Therefore, sending messages to native objects *does not* contribute to *object level coupling*. This is in agreement with [22], as such a message to one of its own components - although this message need not be part of its protocol - will only affect its *own* history as a complex object. However, such a message may well constitute *class level coupling* as discussed above.

As a result, on the object level, *coupling is introduced only by accessing non-native objects*. In the restricted scope of OLC, we can now in principle agree with [4] on the definition of coupling, restating it as the following lemma used to actually detect OLC in a system:

Lemma: Any evidence of a method of one object using methods or instance variables of a non-native object constitutes OLC. ■

Paralleling the discussion in the subsection on class level coupling, we identify the following three dimensions influencing the strength of OLC between objects O and X:

[3]From a technical point of view, native objects may either be physically included in the aggregate (as a "value"), or be associated to the aggregate via a pointer (although this kind of implementation is usually preferred for non-native subobjects).

- *Type of access to X*: O may either restrict its access to the interface of X or may refer to at least one instance variable of X.
- *Scope of X*: Being an object non-native to O, in order for O to have access to X may be one of the following [2]:
  1. a parameter to one of O's methods.
  2. a non-native part of O.
  3. a global object.
- *Complexity of interface*: In the case of message passing, it may be useful to consider the number of arguments of the message to X.

C. An initial ordinal metric within the framework presented

In this section, we give an initial partial mapping of the points of the space spanned by our framework onto an ordinal scale. The most common combinations of dimensions are covered, while some of the missing ones are not likely to occur given a minimum of programming discipline.

Table 1 presents class level coupling. Entries correspond to an ordinal measure of the strength of coupling contributed by a single access by a method of C to some server SC.

		SC is stable	SC is unstable	
		Access to interface[4]	Access to interface	Access to implementation
SC is native to CC	CC is a genuine aggregate of SC (SC is the class of an "exclusive" instance variable of CC)	1	3	5
	A local variable of type SC is used within a method of CC	1	2	4
	SC is a superclass of CC	1	3	5
SC is non-native to CC	SC is the class of a "shared" instance variable of CC (pointer or reference)	1	3	5
	SC is the type of a parameter of a method of CC	1	2	4
	CC accesses a global variable of class SC	1	3	5

Table 1: Class level coupling
(Stability × Access Type × Scope of Access)

The specific values assigned to the cells of Table 1 can be motivated as follows:

Strength 1: Accessing the interface of any server class SC, provided SC is a stable class or features at least a stable interface, the most harmless type of class level coupling occurs, as no change dependencies are introduced. Of course, accessing a global variable (line 6 in Table 1) will certainly lead to strong coupling at the object level (cf. Table 2 below).

Strength 2: Changing the interface of an SC method called via an object local to one of CC's methods, only this latter method needs to be changed correspondingly. The same argument applies to the case where SC is the type of a parameter of a CC method.

Strength 3: Changing the interface of an SC method invoked via a message sent to one of CC's instance variables of class SC, due to the class scope of instance variables, potentially all methods of CC are affected. This is why this case is less favorable than the above.

Similarly, changing the interface of a method of the superclass SC of CC affects all methods of CC calling this superclass method. Thus, again potentially all methods of CC may be affected.

As a global variable is accessible from all methods of a class, the same argument applies for global variables, too.

Strengths 4 and 5: Following the same arguments as for strengths 2 and 3 and noticing that change dependencies are generally stronger when breaching the information hiding principle, these assignments result.

Table 2 displays the cases where non-native objects are involved. Here both, class level and object level coupling occur, however, only the OLC strengths are given, while in Table 3 OLC and CLC (where applicable) are combined.

	Access to interface	Access to implementation
SC is the class of a shared instance variable of CC	II	V
SC is the type of parameter of a method of CC	I	IV
CC accesses a global variable of class SC	III	VI

Table 2: Object level coupling
(Access Type × Scope of Access)

Entries correspond to the strength of coupling constituted by a single access by a method of CC to the corresponding SC object, without considering the complexity of the message interface.

[4]We exclude column “Access to implementation” here because we are not yet sure of the respective value assignments. We thus fail to cover all possible cases. We feel that this omission is not very significant, because in many cases, this kind of access is prohibited anyway. However, if it does occur, the coupling strength is allocated somewhere between the values in column 1 and column 3: It is stronger than in the case where only the interface is accessed, because it introduces some dependency on the class invariants of SC (i.e., C cannot be modified freely without taking into account SC’s design). On the other hand, it is for obvious reasons weaker than in the case of unstable classes, because the probability of changes in SC itself is zero or at least very low.

Strength I: Obviously, sending a message to a non-native object passed as a parameter is the cleanest way of communication, clearly documenting the dependency thus established.

Strengths II and III: In accordance to the findings of structured programming, accessing a global variable is inferior to the case above. As the scope of instance variables is smaller than the scope of global variables, but wider than the scope of parameters, the upper left cell is assigned a medium coupling strength of II.

At this point, it seems to be worthwhile to spend some more thoughts on the assignment of strength III to the lower left cell. In the structured programming paradigm, two modules are “common coupled” when they share a common, globally defined storage space. As Budd explains, this kind of coupling can be avoided in object-oriented systems by introducing a new class [3]:

“*In an object-oriented framework, an alternative to global data coupling that is frequently possible is to make a new class that is charged with “managing” the data values, and route all access to the global values through this manager. (This is similar to our use of access functions to shield direct access to local data within an object). This reduces global data coupling to parameter coupling, which is easier to understand and control.*”

Until now, we have used the notion of *object* for instances of all kinds of data types. Here, we must distinguish between basic types of the implementation language like integer, character and so forth from “real” objects. If SC belongs to the former group, it is hard to imagine that a corresponding instance x is considered an “object” with respect to CC, because of the different level of abstraction. In this case, two clients using x are not directly coupled, although there is a strong indirect coupling (via x) in the classical sense. We may subsume this case in the lower *right* cell of Table 2, identifying the access to a “naked” variable x with the case of access to the implementation of some global object. On the other hand, we assigned strength III to the lower *left* cell by considering SC as a genuine class. In comparison to the coupling classification of the traditional paradigm, this case is equivalent to the more desirable “parameter coupling” because (as stated by Budd, cf. above) the global object of type SC stands for a global *module* rather than for a global data item.

Strengths IV, V, and IV: Arguments motivating assignments in the right column are similar to those for the left column given above, however taking into consideration the penalties resulting from violating the information hiding principle.

Combining the compatible parts of Tables 1 and 2 yields Table 3.

At this point, we must recall that the values in Tables 1 and 2 belong to *ordinal scales*, prohibiting actual computation of the sums in Table 3. Moreover, the two scales are also different and thus incomparable; this is why they are given as arabic and roman ordinal numbers, respectively.

However, partitioning Table 3 with respect to the stability dimension, we obtain two tables where the value assignments happen to be totally comparable. We can thus map each them onto 2 distinct new ordinal scales as follows. For the stable

	SC is stable or has a stable interface	SC is unstable	
	Access to interface	Access to interface	Access to implementation
SC is the class of a shared instance variable of CC	1 + II	3 + II	5 + V
SC is the type of parameter of a method of CC	1 + I	2 + I	4 + IV
CC accesses a global variable of class SC	1 + III	3 + III	5 + VI

Table 3: Overall coupling strength (non native server objects)

case, we have 1+I < 1+II < 1+III which leads to Table 4:

	SC is stable or has a stable interface
	Access to interface
SC is the class of a shared instance variable of CC	2
SC is the type of parameter of a method of CC	1
CC accesses a global variable of class SC	3

Table 4: Overall coupling strength for stable (non-native) servers

Table 5 (the entries of which *cannot* be compared with those of Table 4!) can be constructed in a similar manner for the unstable case.

	SC is unstable	
	Access to interface	Access to implementation
SC is the class of a shared instance variable of CC	2	5
SC is the type of parameter of a method of CC	1	4
CC accesses a global variable of class SC	3	6

Table 5: Overall coupling strength for unstable (non-native) servers

Considering also the interface complexity not yet covered by the values of Tables 2 to 5, one could borrow the idea of Fenton [10] to define a fine grained metric in the form of

$X + p / (1+p)$

where X denotes the respective table entry and p is the number of parameters passed.

Summing up, we may conclude that coupling appears to be a complex notion that does not lend itself to be measured by a single measure. Thus, a software engineer interested in coupling should realize which aspect(s) of coupling she is in fact interested in and should consequently concentrate her investigations on suitably defined attributes. In the following section, we present some examples of such coupling-related attributes.

D. Coupling-Related Attributes

Change Dependency Between Classes (CDBC)

As an application of our framework defined above, let us investigate a special case of CLC in the context of unstable server classes. The attribute *change dependency between classes* (CDBC) determines the potential amount of follow-up work to be done when class SC is being modified in the course of some maintenance activity. While the actual number of changes necessary to bring class CC to par is not predictable in general, CDBC determines the number of methods to be *considered* upon such a change of SC.

According to our framework, CDBC depends on

1. the scope of visibility of the changed server class within the client class (determined by the type of relationship between CC and SC) and
2. the kind of access of CC to SC (interface access or implementation access).

Investigating possible relationship types yields Table 6, where n denotes the number of methods of class CC.

	α = number of methods of CC potentially affected by a change
SC is not used by CC at all	0
SC is the class of an instance variable of CC	n
Local variables of type SC are used within j methods of CC	j
SC is a superclass of CC	n
SC is the type of parameter of j methods of CC	j
CC accesses a global variable of class SC	n

Table 6: Relationship types between CC and SC and their corresponding contribution α to change dependency.

Of course, α is only relevant if those parts of SC accessed by CC are in fact subject to change. This motivates the point 2 mentioned above: If SC represents a mature abstraction, its interface is assumed to be much more stable than its implementation. Thus, many of the changes to the implementation of SC can be performed without affecting its interface. We therefore introduce a factor k ($0 \le k \le 1$) corresponding to the stability of SC's interface and multiply the contribution of an access to the interface with 1-k.

We can now express the degree of CDBC as follows:

$$A = \sum_{\substack{\text{accesses i to}\\\text{implementation}}} \alpha_i + (1-k) \cdot \sum_{\substack{\text{accesses i to}\\\text{interface}}} \alpha_i$$

$$\mathrm{CDBC}\,(CC, SC) = \min\,(n, A)$$

CDBC can be minimized by restricting access to the interface of the server class and restricting the visibility of server classes to small scopes, which may be enforced by language mechanisms like the access specifiers in C++ (public, protected, private). Thus, for designs conforming to the usual guidelines for object-oriented design with stable interfaces and access restricted to the interface only, the change dependency becomes negligeable.

We want to point out that although the measure suggested above may be subject to further improvements, the underlying internal attribute itself is certainly highly relevant for controlling the quality of a product in a changing environment, as stressed by Jarke and Pohl [11].

Locality of Data (LD)

Locality of data (LD) represents an attribute directly connected with the quality of the abstraction embodied by a class. Classes with high data locality are more self-sufficient than those with low data locality. This attribute influences external attributes like the class's reuse potential or its testability.

We can construct a measure for LD by relating the amount of data local to the class to the total amount of data used by that class. For C++, we can define more precisely for a class C

$$LD = \frac{\sum_{i=1}^{n} |L_i|}{\sum_{i=1}^{n} |T_i|}$$

with

M_i $(1 \le i \le n)$	methods of class C (excluding all trivial read/write methods for instance variables)
L_i $(1 \le i \le n)$	set of "local" variables accessed by M_i (directly or via read/write methods). These are: non-public instance variables of class C, inherited protected instance variables of its superclasses, static variables defined locally in M_i
T_i $(1 \le i \le n)$	set of *all* variables used in M_i, except for nonstatic local variables defined in M_i

For the sake of robustness of the measure, we excluded all auxiliary variables defined in M_i because they don't play a major role in a design.

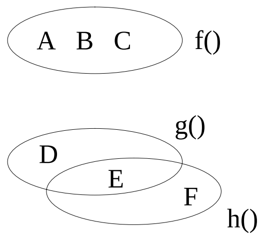
A protected instance variable which has been inherited by a class C is local for an object CO of type C (and hence a member of L_i), even though it is not declared in class C. Using such a variable by methods of C is harmless with respect to LD (it is only accessible by CO itself) but it is undesirable if we are interested to achieve a low value of CDBC. This example shows again that coupling is a multi-dimension attribute which must be divided into more elementary ones[5].

IV. Cohesion

Cohesion is an important attribute corresponding to the quality of the abstraction captured by the class under consideration. Good abstractions typically exhibit high cohesion. The original object-oriented cohesion metric as given by Chidamber and Kemerer [4] (and clarified by the same authors in [5]) represents an inverse measure for cohesion. They define *Lack of Cohesion in Methods* (LCOM) as the number of pairs of methods operating on *disjoint* sets of instance variables, reduced by the number of method pairs acting on at least one *shared* instance variable[6]. The definition given in [5] is reproduced below:

"Consider a Class C_1 with n methods $M_1, M_2, ..., M_n$. Let $\{Ij\}$ = set of instance variables used by Method Mj.
There are n such sets $\{I_1\}, ..., \{I_n\}$.
Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \varnothing\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \varnothing\}$.
If all n sets $\{I_1\}, ..., \{I_n\}$ are $\varnothing$ then let $P = \varnothing$.
LCOM $= |P| - |Q|$. if $|P| > |Q|$
$= 0$ otherwise."

For example, in class X below, there are two pairs of methods



```
class X {
        int A, B, C, D, E, F;
        void f () { ... uses A, B, C ... }
        void g() { ... uses D, E ... }
        void h() { ... uses E, F ... }
};
```

accessing *no* common instance variables (<f, g>, <f, h>), while exactly one pair of methods shares variable E, namely, <g, h>. Therefore, LCOM is 2 - 1 = 1.

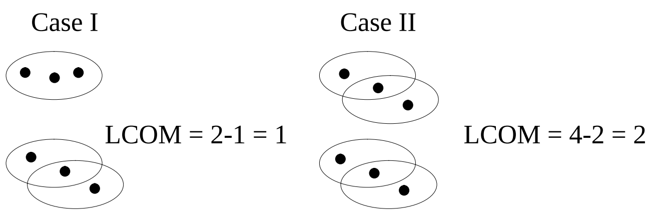
Although the principle idea behind this definition seems very sensible, the resulting cohesion metric exhibits several anomalies with respect to the intuitive understanding[7] of the attribute, the most important of which will be explained below.

Consider the following two designs (each Venn diagram represents a method by the set of instance variables it employs):

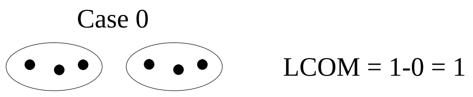
[5]In fact, the first motivation to define LD stemmed from the analysis of a related general coupling measure defined by Stiebellehner [19], who does not distinguish between the distinct aspects of CDBC and LD.

[6]Note that this approach is not applicable to abstract classes due to the lack of method definitions and instance variables.

[7]In the course of preparing the camera ready version of this paper, a report by Eder et al. was published which contains a comprehensive taxonomy of cohesion attributes which includes our view under the term *separable cohesion* [8].



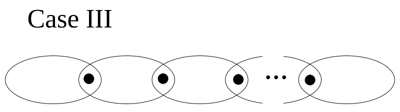
According to our own the empirical relations, both cases are equally non-cohesive: Without taking into account any semantic information, we must conclude that *both* classes should be broken up, despite the lower LCOM-value for Case I. However, if the reader feels that the additional method in Case II should be reflected by the LCOM-score, it should most probably yield a *lower value* rather than a higher one! However, if the reader insists in the correct behavior of LCOM, i.e., in the additional method qualifying for the higher score, then removing one of the overlapping methods of Case I should, *ceteris paribus*, yield a lower value - however, it does remain the same:



Moreover, we dislike the fact that LCOM depends on the number of methods n: Given the fact that the total number of pairs is $\binom{n}{2}$, we conclude that

$$\mathrm{LCOM} = \left\lceil \frac{4|P| - n(n-1)}{2} \right\rceil^{+}$$

where P denotes the set of disjoint (w.r.t. the instance variables used) method pairs and $\lceil k \rceil^{+}$ yields k, if k>0, 0 otherwise. Now, for a family of classes that we deem structurally equivalent, the n in the above formula becomes significant. Consider for example the following general class structure, where n methods are sequentially "linked" by shared instance variables:



Thus we have $|P| = \binom{n}{2} - (n-1)$ yielding

$$\mathrm{LCOM} = \left\lceil \binom{n}{2} - 2(n-1) \right\rceil^{+}$$

For n<5, LCOM is 0, for n=5, 6, 7, and 8, LCOM becomes 2, 5, 9, and 14, respectively, which is certainly counter-intuitive, as it is in fact equally hard (or easy) to split the class in two in each case.

A. Improving LCOM

Li and Henry [13] improved the original version of LCOM given in [4][8] as follows:

"LCOM = number of disjoint sets of local methods; no two sets intersect; any to methods in the same set share at least one local instance variable; ranging from 0[9] to N; where N is a positive integer."

[8]It is interesting to note that the improved version by Chidamber and Kemerer [5] suffers more measure theoretic anomalies than the older improvement by Li and Henry [13].

With this definition, an LCOM value of k>1 hints at the possibility to split X into k smaller and more cohesive classes. Applying this definitions to cases 0, I, and II above yields a value of 2 in all cases which is in accordance with the intuitive notion of cohesion. In all instances of Case III, the value of LCOM is 1, which seems again sensible.

Although with the improved definition of Li and Henry the anomalies discussed above disappear, their set-theoretic formulation is still not quite precise. Hence, we re-state their definition in graph-theoretic terms as follows:

Let X denote a class, I_X the set of its instance variables of X, and M_X the set of its methods. Consider a simple, undirected graph $G_X(V, E)$ with

$V = M_X$ and $E = \{\langle m, n\rangle \in V \times V \mid \exists\, i \in I_X: (m \text{ accesses } i) \wedge (n \text{ accesses } i)\}$.

LCOM(X) is then defined as the number of connected components of G_X ($1 \le \mathrm{LCOM}(X) \le |M_X|$).

In addition to this mere formal improvement of the definition of LCOM, we would like to get rid of a more semantic flaws in the definition of LCOM: Firstly, the not uncommon design principle to restrict accesses to instance variables to special purpose read/write methods introduces an anomaly of this measure: An otherwise cohesive class would yield very high LCOM-values, as all of the "real" methods would yield isolated nodes in the graph, as they do not directly share any instance variable anymore. Secondly, there are many practical cases in which methods exist which do not at all access instance variables (neither directly nor via mere access methods) but are coded entirely in terms of other (more basic) methods of their class. For example, in a class LIST method size() might be recursively defined in C++ as

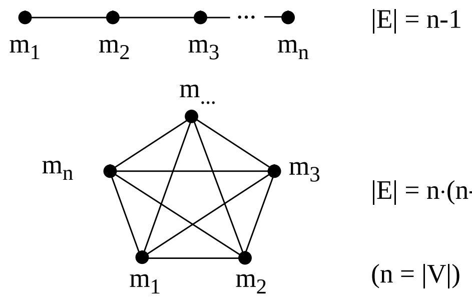
int size() const { return empty() ? 0 : 1 + tail().size(); }

Although such methods can certainly be considered very cohesive, according to the above definitions of LCOM they are classified as "structurally unrelated" to the rest of the class.

To avoid these anomalies, the definition of G_X could be changed as follows:

$E = \{\langle m, n\rangle \in V \times V \mid (\exists\, i \in I_X: (m \text{ accesses } i) \wedge (n \text{ accesses } i)) \vee (m \text{ calls } n) \vee (n \text{ calls } m)\}$

In the cases where LCOM = 1, there are still more and less cohesive classes possible. Especially for big classes, it might be desirable to find a finer grained measure to tell the structural difference between the members of the set of classes with LCOM = 1. For this purpose, let us consider the two extreme cases of connected components:



[9]0 can only occur in cases of classes without any methods, a certainly degenerate case. A more sensible lower value is therefore 1.

The chain represents the minimum cohesive graph with LCOM=1, while maximum cohesion occurs in the complete graph. The obvious generalization of connectivity leads to the graph theoretic notion of "connectivity of degree k" (k edges must be removed to disconnect the graph) which is unfortunately not easy to determine for higher values of k. Instead, we propose a linear mapping of the interval [n-1, n·(n-1)/2] onto the interval [0, 1] as follows:

$$C = 2\frac{|E| - (n-1)}{(n-1) \cdot (n-2)}$$

For classes with more than two methods, C can be used to discriminate among those cases where LCOM=1 as C gives us a measure of the deviation of a given graph from the minimal connective (that is, cohesive) case.

V. Conclusions and future work

Having introduced a framework for a comprehensive metric for coupling in object-oriented systems on both, object and class levels, we were able to identify a basic ordinal metric for the contribution certain elementary constructs provide to coupling.

As an application of the framework, consider the trade-off discussed in [1], namely, if *using* a (non-native) object is preferable to *containing* an object. Denoting the class of such an object by X, we find from Table 1 of our framework that if X is stable, accessing an instance variable of this type X yields coupling strength 1 for the *containing* case. The *using* case is given as 1+I in the second row / first column cell of Table 3. Thus, *containment* is preferable in this case.

Several open problems remain to be solved:

Unifying both sets of values as defined by Table 1 and Table 2 in order to achieve a complete ordinal scale within the coupling framework is of course desirable. To achieve consistent and satisfying results, empirical data obtained from real-life software engineering projects need be analyzed with respect to the influence of the metrics proposed on external product attributes. This applies as well to the cohesion measures presented.

Acknowledgment

The authors would like to thank Günther Vinek for several fruitful discussions.

References

[1] Grady Booch. Object-Oriented Design. Benjamin Cummings, 1991.
[2] Grady Booch. Object-Oriented Design. Second Edition, Benjamin Cummings, 1994.
[3] Timothy A. Budd. An Introduction to object-oriented Programming. Addison Wesley, 1990.
[4] Shyam R. Chidamber, Chris F. Kemerer. Towards a Metrics Suite for object-oriented Design. In Proc. OOPSLA '91, ACM 1991, 197-211.
[5] Shyam R. Chidamber, Chris F. Kemerer. A metrics suite for object-oriented design. IEEE Trans. Software Eng., vol. 20, no. 6, June 1994, 476-493.
[6] Peter Coad, Edward Yourdon. Object-Oriented Design. Yourdon Press, 1991.
[7] Craig Damon, Gordon Landis. Abstract State and Representation in Vbase. In: Object-Oriented Databases With Applications to CASE, NETWORKS and VLSI CAD (ed. Rajiv Gupta, Ellis Horwitz). Prentice Hall, 1991, 178-188.
[8] Johann Eder, Gerti Kappel, Michael Schrefl. Coupling and Cohesion in Object-Oriented Systems.Technical Report, University of Linz, Institut für Informationssysteme, 1995 (submitted for publication).
[9] Lem O. Ejiogu. Five Principles for the Formal Validation of Models of Software Metrics. ACM SIGPLAN Notices, August 1993.
[10] Norman E. Fenton. Software Metrics - A Rigorous Approach. Chapman & Hall, 1992.
[11] M. Jarke, K. Pohl. Requirements engineering in 2001: (virtually) managing a changing reality. Software Engineering Journal, Nov. 1994, 257-266.
[12] Jean Pierre LeJacq. Semantic-Based Design Guidelines for Object-Oriented Programs. JOOP, Focus on Analysis & Design, 1991, 86-97.
[13] W. Li, S. Henry. Maintenance Metrics for the Object Oriented Paradigm. In Proc. 1st Int. Software Metrics Symp., Los Alamitos, CA, May 21-22 1993, IEEE Comp. Soc. Press, 1993, 52-60.
[14] Karl Lieberherr, Ian Holland, Arthur Riel. Object-Oriented Programming: An Objective Sense of Style. In Proc. OOPSLA '88, ACM 1988, 323-334.
[15] Karl Lieberherr, Ian Holland. Assuring Good Style for Object-Oriented Programs. IEEE Software, September 1989, 38-48.
[16] Allen Macro, John Buxton. The Craft of Software Engineering. Addison-Wesley, 1987.
[17] Teri Roberts. Metrics for Object-Oriented Software Development. Workshop Report in Addendum to the Proceedings OOPSLA '92, ACM 1992, 97-100.
[18] W. P. Stevens, G.J. Myers, L. L. Constantine. Structured Design. IBM Systems Journal, Vol. 13, No. 2, May 1974, 115-139.
[19] Johann Stiebellehner. Kopplung in Objektorientierten Systemen - Definition und Bewertung von Kopplungsmaßzahlen. Dissertation, Institut für Angewandte Informatik und Informationssysteme, University of Vienna, 1993
[20] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In Proc. OOPSLA '86, ACM 1986, 84-91.
[21] Douglas A. Troy, Stuart H. Zweben. Measuring the Quality of Structured Designs. Journal of Systems and Software, Vol. 2, 1981.
[22] Yair Wand, Ron Weber. An Ontological Model of an Information System. IEEE Transactions on Software Engineering, Vol. 16, No. 11, November 1990, 1282-1292.
[23] Iseult White. Using the Booch Method - A Rational Approach. Benjamin Cummings, 1994.
[24] Norman Wilde, Ross Huitt. Maintenance Support for Object-Oriented Programs. IEEE Transactions on Software Engineering. Vol. 18, No. 12, December 1992.
[25] Horst Zuse, Peter Bollmann. Software Metrics: Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics. ACM SIGPLAN Notices, August 1989.