Coupling and Cohesion in Object-Oriented Systems

Johann Eder[1] Gerti Kappel[2] Michael Schrefl[3]

[1] Institut für Informatik, Universität Klagenfurt

Universitätsstr. 65, A-9020 Klagenfurt, Austria, email: eder@ifi.uni-klu.ac.at

[2] Institut für Informatik, Universität Linz

Altenbergerstr. 69, A-4040 Linz, Austria, email: gerti@ifs.uni-linz.ac.at

[3] Institut für Witschaftsinformatik, Universität Linz

Altenbergerstr. 69, A-4040 Linz, Austria, email: schrefl@dke.uni-linz.ac.at

Abstract

Object-oriented system development is gaining wide attention both in research environments and in industry. A severe problem encountered, however, is the quickly increasing complexity of such systems and the lack of adequate criteria and guidelines for "good" designs. To cope with this problem, it is imperative to better understand the properties and characteristics of object-oriented systems. In this paper, we extend the concepts of coupling and cohesion developed initially for procedure-oriented systems to object-oriented systems. Coupling describes the interdependency between methods and between object classes, respectively. Cohesion describes the binding of the elements within one method and within one object class, respectively. We introduce a comprehensive taxonomy of coupling and cohesion properties of object-oriented systems and provide guidelines for improving these properties.

CR Classification: D.1.5 [**Software**]: Object-Oriented Programming; D.2.3 [**Software**]: Coding – *standards*; D.2.9 [**Software**]: Management – *software quality assurance*; D.2.10 [**Software**]: Design – *methodologies*

Additional Keywords and Phrases: coupling, cohesion, object-oriented design guidelines

1 Introduction

Building quality systems has been the driving goal of all software engineering efforts within the last two decades. Among the key qualities a system should support are maintainability, extendibility, understandability, and reusability [4, 37]. Object-oriented languages and

development environments go already a long way in providing features for building such maintainable, extendible, understandable, and reusable systems. Among the features which object-oriented languages support are encapsulation and information hiding, user-defined classes, message passing, inheritance, and polymorphism [46]. Among the features which development environments support are predefined class libraries and application frameworks, standardized protocols, e.g., naming conventions, and development tools such as browsers and inspectors [38, 47].

Object-oriented languages and development environments alone, however, are not a panacea for building object-oriented quality systems. For example, one problem encountered is the quickly increasing complexity of object-oriented systems, which is due to the many objects and inter-object relationships in such systems. Hence, what is needed in addition to an object-oriented language and development environment are guidelines, check lists, and metrics which help in the design and development of "good" object-oriented systems - like they were essential for building "good" procedure-oriented systems.

There is an increasing awareness of this problem in the "object community". Several guidelines and methods for object-oriented system development have emerged independently, e.g., [1, 20, 22, 23, 28, 31, 50, 57]. Recently, design metrics which take the idiosyncratic features of object-oriented designs into account have also emerged [2, 9, 10, 18, 19, 41, 42, 43, 54, 55]. A common objective of these design metrics is that they, besides others, try to quantify the coupling and cohesion properties of the object-oriented system under investigation. For example, in [55] the authors identify complexity measures which influence coupling and cohesion of object-oriented systems. Besides this metric-centered view, the notions of coupling and cohesion have been re-discovered in literature as design characteristics influencing properties of object-oriented quality systems [3, 5, 6, 11, 37, 49, 54]. Coupling and cohesion are the primary attributes that led to procedure-oriented quality systems [48, 53]. There, coupling is a measure of the interdependencies between different modules, and cohesion is a measure of the binding of the elements within a single module. To be rated as well-designed, a procedure-oriented system has to have low coupling properties in terms of few interdependencies between modules and high cohesion properties in terms of strong bindings between the elements within a single module. Since those days coupling and cohesion have been adapted to Ada tasking [45] and abstract data type-based system development [14, 15]. As is stated in [14], coupling and cohesion significantly influence maintainability, understandability, and modifiability, and thus serve as a guide to the development of quality systems.

Despite of the growing awareness of coupling and cohesion and to the best of our knowledge, there exists no thorough discussion of coupling and cohesion properties of object-oriented systems in literature. The goal of this paper is to fill this gap. We introduce a comprehensive taxonomy of coupling and cohesion properties of object-oriented systems and provide guidelines for improving these properties. We consciously do not try to quantify the various coupling and cohesion properties besides placing them on some ordinal scale. This is due to the following reasons. Firstly, we do not try to reinvent coupling and cohesion properties from scratch but adapt existing research in the area of procedure-oriented systems and abstract data type-based systems. In that realm, no quantifiable and computable metrics have been investigated. And secondly and most importantly, so far there are no empirical results on the basis of which any coupling and

cohesion metric could be verified. Thus, it would not be serious to give numbers with no experimental backing.

When adapting coupling and cohesion properties from procedure-oriented systems the following considerations are in place. Firstly, as opposed to procedure-oriented systems where the module is the only subject of interest, in object-oriented systems there exist several subjects of interest, such as methods and object classes. Secondly, coupling and cohesion properties of the various subjects of interest are not independent from each other. For example, the coupling properties between methods of different object classes highly influence the coupling properties between these object classes. The enhancement of coupling and cohesion concepts for object-oriented systems is the main contribution of this paper.

The research to be described has been motivated by the project *MooD* (Methods for object-oriented Development) together with SIEMENS Austria, one of Austria's major software development companies. The aim of the project has been to provide a generic object-oriented software life cycle model, which emphasizes software quality assurance and reusability [16, 26].

The paper is organized as follows: In the next section we introduce the subjects of interest in terms of coupling and cohesion characteristics of object-oriented systems. In Section 3 we analyze coupling quality of various relationships between methods and between object classes. Section 4 studies cohesion properties of methods and object classes, and uncovers interdependencies between coupling and cohesion. Throughout Section 3 and Section 4 we elaborate to which extent existing guidelines improve coupling and cohesion quality. Section 5 concludes the paper and points to further research.

2 Basic Concepts of Object-Oriented Systems

To be able to talk about coupling and cohesion of object-oriented systems we have to identify the basic building blocks of such systems and their possible relationships in advance. One of the most widely used definitions of object-oriented languages and systems stems from Peter Wegner [56], which we will use in this paper. There, an object-oriented language should support the basic concepts of an object, an object class, and inheritance.

An *object* consists of a set of instance variables representing the internal state of the object and a set of methods representing the external behavior of the object. To put it in other words, an object is an encapsulation of state and behavior.

An *object class* can be seen as kind of template specifying state and behavior of a set of similar objects, which are created as instances of an object class by some create method during run-time of a program. Object classes are the basic means of object-oriented design and development. The definition of an object class is based on the principle of *encapsulation* following the abstract data type approach and on the principle of *information hiding* distinguishing between visible and hidden parts of an object class' definition. The visible part is called the *specification* or interface of an object class and in general consists of a set of method specifications. The hidden part is called the *implementation* of an object class and consists of the implementation of the methods and the definition

of the instance variables. We do not consider class variables and class methods in this paper.

An object class may be related to other classes by an *inheritance relationship*, in which case it *inherits* instance variables and methods from them. An object class C related to object class C' by an inheritance relationship is called *direct subclass* of C'. An object class C is an *indirect subclass* of C' if there exists some class $\hat{C}$ such that C is a direct subclass of $\hat{C}$ and $\hat{C}$ is a direct or indirect subclass of C'. An object class C is called *subclass* of object class C', if C is a direct or indirect subclass of C'. Conversely, an object class C' is called *superclass* of object class C, if C' is a *subclass* of C. Objects which are instances of subclasses of C are called *members of* C. Inheritance from a single direct superclass is called single inheritance, and inheritance from multiple direct superclasses is called multiple inheritance. The inheritance relationship is transitive, reflexive, and anti-symmetric. The directed acyclic graph built up by the inheritance relationship is called *inheritance hierarchy*. We deliberately use the term inheritance hierarchy instead of the more precise term inheritance graph since it is commonly accepted in the literature.

Besides these basic concepts there are other principles of the object-oriented paradigm, namely message passing and object identity [46], which must be considered when talking about object-oriented quality systems. An object o communicates with an object o' by sending a message to o'. Adhering to the principle of information hiding *message passing* is the only means to access and alter an object's state. Message passing implies the second kind of relationship, the interaction relationship. The interaction relationship is defined for methods in the first place, and deduced for object classes for which the methods are specified in turn. Since the inheritance relationship strongly interferes with the interaction relationship some definitions are necessary before we are able to introduce the interaction relationship.

We first define the notions of overriding, polymorphism, dynamic binding, and static class. Overriding refers to the redefinition of inherited instance variables and methods in subclasses. Polymorphism means that the same method may be invoked on objects of different classes. Dynamic binding means that the binding between method invocation and code to be executed takes place during run-time and depends on the actual class of the object on which the method is invoked. The static class of a variable is the domain of this variable defined at compile time. In contrast, the dynamic class of a variable is the actual class of some object referenced by the variable during run-time. Due to polymorphism a variable with the static class being object class C may reference members of C during run-time. Due to overriding a method may have several signatures and implementations, at most one of each for the class where it has been initially defined, and at most one of each for every subclass. The signature of a method consists of the name of the method, the names and domains of the input parameters, and the domain of the return value. Changing a signature in some subclass comes up to changing the names, domains, and the number of its parameters. We say, a method m *is implemented* at object class C if it is initially defined at C, or its signature and/or implementation have been overridden at C. We further say that a method m *is defined* at object class C if it is implemented at C or at one of the superclasses of C. Concerning the interaction relationship of a class C with other object classes we restrict our attention to those methods of C which are implemented at C. Thus, an object class C is related to another object class C' by an

interaction relationship concerning the methods m implemented at C and m' implemented at C' if in the implementation of m at C the method m' can be invoked on an object referenced by a variable whose static class is either C', or a superclass of C', or a subclass of C' and m is neither implemented at this subclass nor at any class in the superclass chain between this subclass and C'.

Some explanations are in place. (1) We consider only direct interactions at first. The transitive closure of the interaction relationship due to transitive method invocations will be treated in section 3.4. (2) Our definition of interaction relationship takes static *and* dynamic classes into account. Of course, the encountered interaction relationships are potential relationships, which might occur but need not occur during run-time. This is in line with the initial incentive that all possible relationships, and thus all possible dependencies have to be investigated. (3) As a special case of the definition of interaction relationship such a relationship may also exist between object classes which inherit from each other. For example, a method m is implemented at C, a method m' is implemented at C', and C is a subclass of C'. Then, C is related to C' by an interaction relationship concerning m and m' if in the implementation of m at C the method m' is invoked on an object referenced by a variable whose static class is either C', or a superclass of C', or a subclass of C' and a superclass of C and m is not implemented at any class in the superclass chain between this subclass and C'. (4) The interaction relationship has been defined for object classes and methods together. For sake of simplicity, however, the interaction relationship and thus interaction coupling will be discussed for methods in the first place, and in a second step extended to object classes (cf. Section 3.1 and 3.4).

The third relationship which is relevant in terms of coupling properties is made possible by the concept of *object identity*, which means that each object has a unique system-maintained identifier, which does not change in time. Thus it is possible that an object may reference other objects via its instance variables using the objects' identifiers. These objects again may reference other objects and so on. An object class C is related to another object class C' by a *component relationship* if C' is used as domain of some instance variable of C. Note, for the purpose of investigating coupling properties there is no need to distinguish between componenent relationships in the more restricted sense, such as part hierarchies, and component relationships in the general sense, such as general references between independent objects. Contrary to the interaction relationship, we consider only direct component relationships in this paper.

Based on the above analysis *methods and object classes* are identified as *basic building blocks* when considering coupling and cohesion of object-oriented systems. Methods are the basic means of invocation and thus, most similar to modules in procedure-oriented systems. Note, in this paper the term *module* is used as in the original literature on coupling and cohesion [53, 59] synonymous to procedure, subroutine, or similar programming units. Object classes are the basic units of encapsulation and thus, basic building blocks by definition. In contrast, we do not consider objects and their coupling and cohesion characteristics. Remember that coupling and cohesion are primarily investigated in the realm of object-oriented design, which implies modeling of object classes but not of individual objects. Furthermore, it is not necessary to consider more complex building blocks such as subsystems [58]. A subsystem consists of a set of object classes, which cooperate to fulfill a certain functionality. Subsystems can be seen as more complex object classes,

usually without inheritance. Thus the discussion of coupling and cohesion characteristics of object classes can be easily extended to subsystems by recursively applying the rules defined for classes.

In the following sections we will deal with methods, object classes, and instance variables. Whereas methods and object classes are the primary subjects of interest in terms of coupling and cohesion properties, instance variables are necessary means to exhibit these properties. We will use C++-style syntax [13] in our examples.

3 Coupling

Coupling has been defined the first time in the realm of procedure-oriented systems [53]. Stevens et al. define coupling as "the measure of the strength of association established by a connection of one module to another. Strong coupling complicates a system, since a module is harder to understand, change, or correct by itself if it is highly interrelated by other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules". In the previous section we defined as the object-oriented equivalent of a module a method. Similar to modules, methods are coupled by invocation of each other and/or by sharing data. Thus they may have an interaction relationship with each other (cf. Section 2). Next to methods, also object classes have to be analyzed in terms of relationships with each other, and thus in terms of coupling properties. Object classes may have component relationships and inheritance relationships with each other, in addition to interaction relationships (cf. also Section 2). From these relationships three different dimensions of coupling properties may be deduced in object-oriented systems, which are:

- interaction coupling
- component coupling
- inheritance coupling

Each of these coupling dimensions induces that the behavior of a class C depends on the behavior of a class C' if C is related to C' by one of the relationships mentioned before. Or, to put it in other words, C has to have some information about C' such that in case C' changes, C knows what to change, too. The *degree of coupling* can be described as *how much*, *how complex* and *how explicit* this information has to be. On one end of the scale, low coupling is described by a small, simple and explicit inter-relationship between methods and between object classes. In general, low coupling correlates to good software quality in terms of better maintainability and reusability. On the other end, high coupling is described by a large, complex and implicit inter-relationship making maintenance a nightmare and reuse even impossible. In the following we study each of the coupling dimensions in turn.

3.1 Interaction Coupling

Methods are coupled by interaction in terms of invocation of each other and/or sharing of data. Since interaction coupling is most similar to the classical definition of coupling between modules we adopt the various degrees of classical coupling [53, 59] to describe interaction coupling. In the following we analyze where interaction coupling in object-oriented systems differs from the classical notion of coupling. The difference mainly stems from two interrelated facts. Firstly, methods belong to object classes. This implies that object classes may be interaction coupled, too. Secondly, interaction coupled methods may belong to the same object class. This implies that we have to distinguish interaction between different classes from interaction within a single class.

We consider all degrees of interaction coupling in turn - from worst to best - and redefine them according to the idiosyncracy of object-oriented systems where necessary.

1. **content**

   Content coupling is the worst form of coupling. It means that one method directly accesses parts of the internal structure, i.e., the implementation of another method. Thus one method has to know exactly all internals of the other methods, and any change in one method may influence the other.

   The object-oriented paradigm in general, and encapsulation and information hiding in particular prohibit that a method directly accesses the implementation of another method or hidden instance variables of a different class. However, content coupling may occur if the programmer uses features of some object-oriented languages which break the information hiding property. For example, the friend option in C++ [13] allows to access hidden, i.e., private or protected in C++ parlance, instance variables of different object classes.

2. **common**

   Coupling is rated common if methods communicate via an unstructured, global, shared data space. Common coupling is better than content coupling since all implicit communication channels are collected in the common area. Nevertheless, it is still a pathological form of coupling since the number of possible connections between methods is polynomial, and the locality principle of good software design is not considered at all.

   Encapsulation and information hiding prohibit common coupling. We are also not aware of any object-oriented language which supports an unstructured, globally visible data space. We rate coupling based on the use of pool variables in Smalltalk [17] as external (see below) since these variables provide for a structured, shared data space with varying visibility.

3. **external**

   External coupling improves common coupling by structuring the global, shared data space. However, the locality principle is still violated, thus most deficiencies of common coupling remain.

Encapsulation and information hiding also prohibit external coupling between methods of different classes. Nevertheless, it may occur in object-oriented systems based on languages which provide globally visible variables. For example, public instance variables in C++ [13] and Trellis/Owl [51], and pool variables in Smalltalk [17].

And what happens to methods of the same class? We may find external coupling in the interaction between methods of the same class as they may access the same instance variables which are used similar to global variables in modules. Passing of data may be implemented through these shared instance variables instead of using explicit parameters. Note, in general, we do not consider the passing of information between different invocations of methods of the same object in instance variables as external coupling. That's what instance variables have been invented for. Coupling is rated as external, however, if instance variables do not represent the state of the object. Such instance variables contain transient data, i.e., data which is only relevant during the execution of a method and is not relevant at the next invocation of a method. Such data may be reinitialized at each invocation from outside. Transient data should be represented by local variables of the method and passed to other methods as parameters. Like global data in modules transient data in instance variables should be avoided. We define the set of instance variables of an object class as being minimal if and only if they contain data representing the (static) state of an object of that class. Methods of such a class cannot be external coupled to each other. Note, methods of different classes, which also do not inherit from each other, may use public instance variables of these classes for passing transient data, which is an even worse form of external coupling. Both kinds of external coupling can and should be avoided.

Example: Consider the definition of an object class EMPLOYEE (note, we assume that the object classes STRING and DATE have been defined elsewhere):

```
class EMPLOYEE {
      STRING* name;              /* all instance variables are private */
      STRING* address;
      int socialSecurityNumber;
      DATE* birthDate;
      int age;
      DATE* hireDate;
      void computeAge ();        /* private method */
public;
      float computeSalary ();
};

void computeAge () {
      age = today - birthDate
};

float computeSalary () {
      ...
};
```

The method computeSalary, which has to know the age of the employee to compute his or her salary, has to call the private method computeAge before accessing the instance variable age. Thus, the instance variable age represents transient data and the calling method is external coupled to the method computeAge. To avoid external coupling computeAge has to return the computed age as return value to the calling method. □

For external coupled methods we further distinguish methods and instance variables implemented at the same class from those implemented at a classes C and a superclass C' of C. We define the *coupling* from method m to method m' as *inherited external*

- if m implemented at C and m' defined at C exchange data through instance variables inherited from C' (instead through explicit parameters). This kind of coupling particularly occurs if the class from which information is inherited has external coupled methods.
- if m implemented at C'' and m' defined at C, communicate via public instance variables inherited by C from C', where $C \neq C''$ and C is neither a superclass nor a subclass of C''.

It is obvious that inherited external coupling between methods is worse than external coupling between methods as it further complicates maintenance. Since inherited variables are directly accessed, inherited external coupling not only uses instance variables to pass transient data but it also breaks encapsulation and information hiding between an object class and its superclasses [52].

Example: Assume for the example above that the object class EMPLOYEE is a subclass of PERSON, from which it inherits the instance variables name, birthDate, age and the method computeAge. Then the coupling from method computeSalary defined at EMPLOYEE to method computeAge defined at PERSON is rated inherited external since they communicate via the inherited instance variable age. Thus the coupling from computeSalary to computeAge is even worse than in the previous example. □

4. **control**

Methods are control coupled if they communicate exclusively via parameter passing, which implies that they are not content, common, or external coupled, but one method controls the internal logic of the other method. With control inversion, the worst form of control coupling, the called method determines the future execution sequence of the calling method.

Control coupling is not prohibited by object-oriented concepts. Therefore, interaction between methods of the same as well as of different classes may be control coupled. Control coupling should be avoided since the change of the implementation of a method may cause hidden changes to the behavior of the control coupled methods. Although control coupling is not prohibited by the object-oriented paradigm, polymorphism and dynamic binding aid in avoiding control coupling. Instead

of passing a flag which controls method execution, polymorphism and dynamic binding can be employed.

5. **stamp**

   Two methods are stamp coupled if, in analogy to classical coupling, they are not control coupled but whole data structures are passed as parameters although only parts of the data structure would suffice. The essence of stamp coupling is as follows: a method depends on some externally defined data structure and has to be changed if this data structure changes, although the change would otherwise not influence the method. Stamp coupling has to be rephrased for object-oriented systems since there exist two kinds of stamp coupling.

   The *first kind* of stamp coupling is similar to the classical definition of stamp coupling. According to that definition, a method depends on the domain of its parameters. The domain of a parameter may either be an object class, or a basic data type, or a complex data type based on type constructors such as tuple, array, and set. Depending on the domain of the parameter, either basic data values, or complex data values, or objects may be passed as parameters. If a complex data value is passed as parameter stamp coupling occurs if already parts of the complex data value would suffice. This case is analogous to the classical definition of stamp coupling. If objects are passed as parameters a similar problem may occur. We recall that an object may again consist of (references to) other objects. Such an object is also called composite object since it is constructed out of component objects. Thus we have to investigate the question whether the object passed as parameter or merely some of its components are relevant for a method. If an object is passed, and the method uses just some of the object's components but not the object itself, we classify this interaction as stamp coupled. If the object passed as parameter is used as a whole, we call it data coupling (see below).

   To improve stamp coupling to data coupling an object should be replaced by its components whenever possible, in particular, if only some but not all of its components are necessary. Note, that there are rare cases where the replacement of an object by its components may leave extensibility more difficult. Such a situation occurs if only some components of an object are currently needed by some method m but the object is extended with an additional component in the future and this component is also requested by m. If the object would have been passed as parameter no change of the interface of m would have been necessary. However, there exist other solutions without the above mentioned problem. For example, adding a new method with the appropriate parameters is just one possible solution.

   Example: Consider the class EMPLOYEE as defined above with the additional instance variable sales, and another object class SALES-STATISTICS with the instance variable accummulatedSales, and the method addSale with the input parameter being an employee object. However, the method addSale should not take an employee object as parameter, which leads to stamp coupling, but the value of the relevant instance variable sales of a particular employee, which would lead to data coupling. □

The *second kind* of stamp coupling uncovers dependencies between a method and the domain of instance variables of the same class. The definition of instance variables is external for a method. At first sight it may look strange to consider interaction between methods and instance variables. However, it leads to rules for a better organization of methods. The value of an instance variable is either a basic data value, or a complex data value, or a reference to some object depending on the domain of the instance variable. If a method directly accesses an instance variable although it needs only parts of its value, the method has to be changed if the domain of this instance variable is changed, e.g., due to optimization purposes.

The key idea for improving this kind of stamp coupling is to distinguish between methods which directly access instance variables, i.e., the internal data structure of a class, and methods which do not. It is good design to hide the internal data structure whenever possible – not only from the outside of an object class but also from methods inside of the object class. Therefore, we suggest to design read methods and write methods, called access methods, for each instance variable and use these methods as only means to access these variables. If the internal data structure of an object class is changed only the access methods have to be updated, too.

Example: Consider the class MATRIX with the methods accessElement and multiplication. The coupling between the method multiplication and the implementation of the matrix's data structure is lower if the method accesses the elements of the matrix by the access operation accessElement, and it is higher if multiplication directly accesses the instance variables. Considering the former, if the representation of the matrix is changed, e.g., for sparse matrices, only the access methods have to be changed but not the multiplication method. □

The idea of restricting the access to instance variables via explicit access methods is not new. It has already been advocated as important object-oriented design guideline [23, 57], and it is realized in some object-oriented languages, such as Trellis/Owl [51]. Note, that there exist design rules in the area of software reuse [40] leading to a factoring out of methods from some object class if they do not directly access instance variables of that class. This might increase, however, interaction coupling between classes. Thus there is some trade-off between various design goals and the designer has to decide which goal to prefer on a case by case basis.

Here again we have to take inheritance of methods and instance variables into account. Stamp coupling between a method and inherited instance variables is called *inherited stamp coupling*. It is worse than stamp coupling between a method and instance variables defined within the same class. This is due to the commonly accepted understanding that directly accessing inherited instance variables in subclasses breaks encapsulation and information hiding [52].

6. **data**

Two methods are data coupled if they communicate only by parameters and these parameters are relevant as a whole. Data coupling is the best form of coupling

whenever two methods have to interact. Data coupled methods minimize maintenance effort due to a great restriction of change propagations.

7. **no direct coupling**

   The theoretical optimum of interaction coupling is no direct coupling, i.e., two methods do not (directly) depend on each other, and thus also their object classes are not interaction coupled. A change in one method does not directly demand a change in the other method, and hence no change in that method's object class is necessary.

3.2 Component coupling

As opposed to interaction coupling where object classes *and* methods are involved component coupling concerns only object classes. The component relationship between classes is defined by the use of a class as domain of some instance variable of another class (cf. Section 2). In the context of component coupling we extend this notion of component and define the object class C' to be a component of the object class C if and only if C' appears in C. C' appears in C if and only if:

1. C' is the domain of an instance variable of C, or
2. C' is the domain of a parameter (input or output) of a method of C, or
3. C' is the domain of a local variable of some method of C, or
4. C' is the domain of a parameter (input or output) of some method invoked within a method of C.

Whereas component coupling reveals the coupling from a class C to a class C' during compile time it might happen during run-time that C is component coupled with any subclass of C'. We call this kind of coupling potential coupling. If C is component coupled with C' then C is *potentially component coupled* with all subclasses of C'. Note, component coupling in languages like Smalltalk needs some special consideration[1]. Since primitives in Smalltalk like integers and booleans are also object classes practically every class is component coupled to primitives. However, since primitives are very stable one shouldn't care too much that an object class is component coupled and thus dependent on primitives.

Of course, component coupling usually implies interaction coupling. In interaction coupling, however, we focused on *how much* information is exchanged between methods and classes, respectively, and on *how complex* this information is. With component coupling we will analyze *how explicit* the coupling between classes is.

The first case of component relationship given above is realized via instance variables. It is made explicit in object-oriented languages at the class level, but only in the implementation part and not in the specification part of a class definition. Coupling of the

[1] We are grateful to David Monarchi who pointed this out to us.

second case is made explicit in the specification part of a class definition by specifying the signatures of the methods. Component coupling of the third case is based on local variables. It is only explicit within a method through the declaration of local variables but it is not explicit at the class level. The fourth case is even worse. For example, in cascading messages, the object returned by a method is used immediately as receiver of another message. The object class of this receiver might not even be declared anywhere in the actual class.

Based on these considerations we define the following degrees of component coupling from worst (highest) to best (lowest).

1. **hidden**

   The coupling between two classes C and C' is rated *hidden* if C' shows up neither in the specification nor in the implementation of C, although an object of C' is used in the implementation of a method of C.

   To give examples of situations where hidden coupling is likely to occur we refer to the cascading message problem stated above. A similar problem is encountered if the return value of a method invocation is immediately used as input parameter in another method invocation. Most languages do not require that the class of this object is declared anywhere within the actual class.

   Hidden coupling causes problems since this coupling between classes is implicit. We compare hidden coupling with the use of global variables in procedure-oriented systems, which is responsible for common coupling between modules. Consider a change of a class in a maintenance process, e.g., the change of the signature of a method. In the presence of hidden coupling the programmer has to search through all implementations of all methods of all classes to detect where this change may have influence, and where this change has to be propagated to, respectively.

   A possibility to avoid hidden coupling is to disallow the use of cascading messages, for example, suggested implicitly by the Law of Demeter [30, 31], and to disallow the use of return values as parameters if their domains are not declared. A less restrictive way to overcome hidden coupling is to declare all those classes in the specification part of the actual class definition.

   Example: Consider the class EMPLOYEE as defined above with the additional instance variable involvedInProject, which references the project for which an employee is currently working, and the additional method numberColleagues, which returns the number of colleagues in the current project. The implementation of numberColleagues may be given as follows:

   ```
   int numberColleagues () {
       return (involvedInProject->getProjectMembers->count - 1)
   }
   ```

   The coupling between the classes EMPLOYEE and SET⟨EMPLOYEE*⟩ is hidden since the latter neither shows up in the specification nor in the implementation of the former although the method count is invoked on an object of

class SET⟨EMPLOYEE*⟩. This object is returned by the method getProjectMembers invoked on the project object which is referenced by the instance variable involvedInProject. Improvements of this implementation in terms of coupling properties are demonstrated in the next examples. □

2. **scattered**

We rate two classes C and C' as scattered coupled, if C' is used as domain in the definition of some local variable or instance variable in the implementation of C yet C' is not included in the specification of C. To detect whether C and C' are component coupled it is necessary to check the implementation of classes to get the domains of instance variables, and even worse to check the implementation of all methods to detect the domains of local variables. If a class is changed the implementations of all other classes have to be checked in order to discover which classes may be influenced.

Example: Consider the previous example where the classes EMPLOYEE and SET⟨EMPLOYEE*⟩ are hidden coupled due to the implementation of the method numberColleagues. The implementation may be improved by disallowing cascading messages as follows:

```
int numberColleagues () {
    SET⟨EMPLOYEE*⟩ * projectMembers;
    projectMembers = involvedInProject->getProjectMembers;
    return (projectMembers->count - 1)
}
```

By introducing local variables and disallowing cascading messages the coupling between the classes EMPLOYEE and SET⟨EMPLOYEE*⟩ can be improved from hidden to scattered. □

3. **specified**

We rate two classes C and C' as specified coupled if C' is included in the specification of C whenever it is a component of C. Specified coupling overcomes the problems of hidden and scattered coupling by specifying all related component classes of some class in a single place. Thus it is possible to determine whether two classes are coupled without browsing through the whole implementation. Browsing the implementation might be even impossible if the source code is not available.

In most object-oriented languages only the signatures of the methods provided by some class C are shown in the specification of C. This set of methods provided by C is also called *suffered interface* of C. Those classes which are used as domains of input parameters and return values of the methods of C are the only ones which are specified coupled with C. We suggest that in addition to the suffered interface also the required interface becomes part of the specification of a class. The *required interface* of some class C comprises all classes which are used as components of C. Specifying which methods of the component classes are invoked further narrows the required interface and thus lowers the degree of coupling.

Example: In the previous example the classes EMPLOYEE and SET⟨EMPLOYEE*⟩ are scattered coupled. We may improve their coupling property to specified coupling by changing the specification of EMPLOYEE as follows:

```
class EMPLOYEE {
suffered interface:              /* corresponds to public in C++ */
      int computeSalary ();
      int numberColleagues ();
      ...
required interface:              /* not available in C++ */
      SET⟨EMPLOYEE*⟩* class PROJECT::getProjectMembers ();
      int class SET⟨EMPLOYEE*⟩::count ();
      ...
};
```

□

There exist several object-oriented specification languages which provide mechanisms to specify the required interface, e.g., collaborators in [58], uses relationship in [5, 21], calling relationships in [24], and invocation diagrams in [25]. However, we know of only one object-oriented language, Modula-3 [8], which supports suffered and required interface specifications by export and import declarations at the module level.

4. **nil**

   The theoretical optimum is no direct component coupling between classes and thus no interaction coupling. It is an advantage to recognize that two classes are completely independent such that one class can be maintained without any knowledge of the other class.

3.3 Inheritance coupling

Similar to component coupling inheritance coupling only concerns object classes. Two classes are *inheritance coupled* if one class is a direct or indirect subclass of the other. Inheritance is one of the most important features of object-oriented methods and languages. It supports reuse both through subclassing, i.e., specialization, and through factoring out, i.e., generalization, of common information from independent classes into a common superclass. At a first glance it seems contradictory to use inheritance for gaining better reusability and to have the goal of low coupling. The key idea to resolve this seeming contradiction is twofold. Firstly, inheritance may be used to lower coupling in an object-oriented system through factoring out. Given a class D which invokes the same method m on objects of class C' and C'', D is component coupled with C' and C''. If the method m is factored out into a common superclass C of C' and C'' and not overridden in C' and C'', respectively, D is component coupled with C only. Thus the coupling is improved since the number of classes with which D is coupled has been reduced. Secondly, there exist different degrees of inheritance coupling. We will show that the lowest degree of inheritance

coupling (besides no coupling) coincides with better reusability. Furthermore, considering inheritance coupling is necessary for improving the overall quality of the system to be implemented. Since it is possible to gain good interaction and component coupling properties by sacrificing the quality of the inheritance hierarchy, inheritance coupling has to be considered, too.

Inheritance coupling is also different from interaction coupling and component coupling in that it does not only exhibit the coupling property between subclasses and superclasses but implicitly also the coupling property between an interaction coupled object class and the inheritance hierarchy. The meaning is the following: if class D is interaction coupled to some class C being the root of an inheritance hierarchy and the inheritance hierarchy is changed, e.g., subclasses are added, and inherited instance variables and methods are modified, the degree of inheritance coupling reveals to which extent changes in the inheritance hierarchy might impose changes in D (for a detailed discussion of the interdependencies of the various coupling dimensions see Section 3.4).

In the following we discuss the various degrees of inheritance coupling from worst (highest) to best (lowest). Note, we assume for simplicity and without loss of generality that access methods for each instance variable exist. Thus any change to an instance variable in some subclass is reflected by the corresponding change of the signature and/or implementation of the access methods.

1. **modification**

   Modification coupling is the worst case of inheritance coupling since in addition to defining new information the inherited information is changed arbitrarily or is even deleted. Depending on the kind of modification we further distinguish between signature modification and implementation modification:

   (a) **signature modification**

   The coupling between subclass C' and superclass C is rated signature modification if not only the implementation but also the signature of an inherited method is changed without any restriction, or inherited methods are deleted in C'. The relationship between C' and C is a pure implementation relationship, i.e., the use of inheritance is solely for code reuse. An inheritance hierarchy based on signature modification is difficult to maintain and to extend since it soon may become very complex. What counts even worse is the fact that it impairs polymorphism and strong typing in interaction related classes. Assume class D which invokes a method m on an object of class C. Assume furthermore that the signature of m is changed in subclass C' of C arbitrarily. Due to polymorphic variables it might happen at run-time that D invokes m on an object of class C'. This object, however, assumes another invocation of m due to signature modification and issues a run-time type error. We conclude that inheritance coupling based on signature modification should be avoided in any case.

   If only a part of the class definition of class C is to be used in class C' the two classes should not be inheritance coupled but C' should be component coupled

to C via an instance variable of C' with the domain of the instance variable being class C.

Example: Consider class STACK inheriting from class ARRAY. Since ARRAY is only used to implement STACK's internal data structure, and since the methods of ARRAY are semantically not meaningful when used with a stack (e.g., the method putAt of ARRAY does not exist for a stack) the methods of ARRAY are only inherited for private use but are deleted from the suffered, i.e., public interface of STACK. Thus STACK and ARRAY are signature modification coupled. To improve their coupling the definition of STACK should include an instance variable a with domain ARRAY instead of inheriting from ARRAY. □

(b) **implementation modification**

The coupling between subclass C' and superclass C is rated implementation modification if the implementation of an inherited method is changed without any restriction. This degree of inheritance coupling is better than the previous one since neither the signature of a method is changed arbitrarily nor are methods deleted. Nevertheless, implementation modification coupling should also be avoided since the semantics of a method may be changed completely in subclasses, and thus the semantics of methods invoking the inherited method may be changed implicitly, too.

2. **refinement**

Refinement coupling is much better than modification coupling since in addition to defining new information the inherited information is only changed due to predefined rules. Depending on the kind of refinement we further distinguish between signature refinement and implementation refinement:

(a) **signature refinement**

The coupling between subclass C' and superclass C is rated signature refinement if they are not modification coupled and if the signature of at least one inherited method m is changed in C' due to some predefined rule without changing the intended semantics of m. Signature refinement may adhere to the covariant rule or to the contravariant rule [7, 34] of subclassing. In the covariant rule, domains of input parameters and of the return value may be replaced by subclasses. In the contravariant rule, domains of input parameters may be replaced by superclasses and the domain of the return value may be replaced by a subclass. In general, signature refinement based on the covariant style should be avoided since it may also break polymorphism and strong typing in interaction related classes [34]. However, there are situations where the semantics of the problem domain is best described in terms of inheritance based on the covariant style. In such situations the polymorphic use of variables and methods should be avoided to avoid run-time type errors.

Example: Consider parts of the definition of object class PERSON and of subclass EMPLOYEE of PERSON:

```
class PERSON {
      [0..120] age;            /* for simplicity we assume */
      ...                      /* the existence of an enumeration type [0..120] */
public;                        /* and [15..65] */
      [0..120] getAge ();
      void setAge ([0..120] a);
      ...
}

class EMPLOYEE : public PERSON {
      [15..65] age;
      ...
public;
      [15..65] getAge ();
      void setAge ([15..65] a);
      ...
}
```

Since employees may only be active from 15 to 65 (at least in Austria) the subclass EMPLOYEE of class PERSON refines the signatures of the inherited access operations of age according to the covariant style. Thus, EMPLOYEE and PERSON are signature refinement coupled based on the covariant style. □

(b) **implementation refinement**

The coupling between subclass C' and superclass C is rated implementation refinement if the signatures of the inherited methods are not changed at all, and the implementation of at least one inherited method m is changed in C' due to some predefined rules such that the intended semantics of m is kept.

This kind of inheritance coupling might become necessary if the implementation of an inherited method has to be refined in some subclass. Language features which support implementation refinement are, for example, SUPER of the language Smalltalk [17], inner of the language Beta [29], and before and after daemons of the language CLOS [27]. SUPER is used in the changed implementation of an inherited method to invoke the method's implementation defined in the superclass. Whereas the concept SUPER does not enforce implementation refinement the inner concept and the before and after daemons do enforce it. inner is used in the original implementation of some method m to specify the place in the code where a future refinement of m has to be placed by the compiler. Thus, if m is invoked on an object of some subclass not only the most refined code but also the code defined in the superclasses gets executed. Similar holds true for before and after daemons. As the name already suggests, they may be specified in subclasses to refine the implementation of m given in some superclass. If m is invoked on an object of some subclass all before daemons get executed up to the original implementation of m and before the original implementation is executed. After the original implementation has been executed all after daemons get executed in the reverse order of the before

daemons.

Note, there exist object-oriented languages, such as Eiffel [39], where it is possible to change the implementation of a method arbitrarily, and at the same time to refine the signature of the same or another method. These languages exhibit both implementation modification coupling and signature refinement coupling.

3. **extension**

   Inheritance coupling between a subclass and its superclass is rated extension coupled if the subclass only adds methods and instance variables but neither modifies nor refines any of the inherited ones. Extension coupling is the best kind of inheritance coupling (besides no inheritance coupling at all). Extension coupling is achieved if the superclass is semantically a generalization of its subclasses and the methods of the superclass can be invoked on objects of the subclasses without inspecting the (intermediate) subclasses.

   Example: Assume that EMPLOYEE is an extension coupled subclass of PERSON. Then, all methods defined at PERSON can be used for employee objects without checking whether they have been modified or refined in the definition of EMPLOYEE. □

4. **nil**

   If there is no inheritance relationship between two classes their inheritance coupling is rated nil.

3.4 Interplay of the three coupling dimensions

So far, we have investigated interaction coupling, component coupling, and inheritance coupling in isolation. In this subsection we reveal the interplay of the three coupling dimensions. In particular, we show by means of an illustrative example how additional coupling relationships are derived from given ones taking transitive method invocations into account.

We recall that interaction coupling and component coupling describe a similar phenomenon. Interaction coupling reveals the kind of interaction between methods and object classes. Component coupling investigates how explicit this interaction is specified. In terms of the interplay with inheritance coupling it is thus sufficient to choose one out of interaction coupling or component coupling to be investigated in more detail. We have chosen interaction coupling.

The interplay of interaction coupling and inheritance coupling becomes most relevant during run-time. Due to inheritance, overriding, and polymorphic variables additional interaction couplings between methods and between object classes, respectively, are derived. For computing all derived couplings during compile time a global analysis of the given code is required, a feasible but tedious and cumbersome task. However, we will show the relevance of such an analysis by means of an example further below.

For a precise description of the problem, we introduce the following three *predicates*:

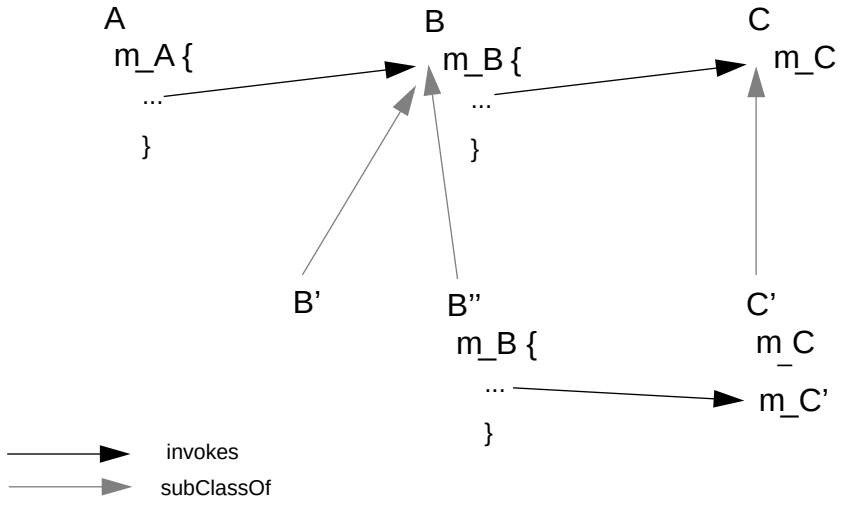

Figure 1: Inheritance relationship and interaction relationship

1. The predicate $implements(C, m)$ holds if class C implements method m. The implementation may define a new method or override an inherited method.

2. The predicate $isa(C', C)$ holds if C' is a direct subclass of C. The predicate $isa^+(C', C)$ denotes the transitive closure, and the predicate $isa^*(C', C)$ denotes the transitive and reflexive closure of $isa(C', C)$.

3. The predicate $invokes(C, m, C', m')$ is true, if method m implemented at C *invokes* method m' on an object *referenced* by a variable with static class C'.

We have defined direct interaction relationship informally in Section 2. Now, we give a formal definition based on the predicates *implements*, *isa*, and *invokes*. Note, *icw* stands for *interaction-coupled-with*.

The predicate $icw(C, m, C', m')$ between object class C and its method m and object class C' and its method m' holds, if there exists a class $\hat{C}'$ such that $invokes(C, m, \hat{C}', m')$ holds, and $implements(C', m')$ holds, and either
(i) $isa^*(C', \hat{C}')$ holds, or
(ii) $isa^+(\hat{C}', C')$ holds and for all $\bar{C}'$, such that $isa^+(\hat{C}', \bar{C}')$ and $isa^+(\bar{C}', C')$ hold, the predicate $implements(\bar{C}', m')$ does not hold.

An indirect interaction relationship between methods m and m' of the classes C and C' holds, if m' implemented at C' may be invoked by some method m'' which has been invoked by method m, implemented at C. We extend the predicate *icw* in that respect.

The predicate $icw(C, m, C', m')$ holds if there exists a class C'' such that $icw(C, m, C'', m'')$ and $icw(C'', m'', C', m')$ hold.

To demonstrate the applicability and usefulness of the introduced predicates consider the example in Figure 1. It depicts the inheritance relationships of the object classes A, B, B', B'', C, and C' as well as their interaction relationships due to method invocations. All coupling predicates which may be deduced from Figure 1 are summarized in Table 1.

From − To	Predicates & Derivations
	$implements(A, m\_A)$
	$implements(B, m\_B)$
	$implements(C, m\_C)$
	$implements(B'', m\_B)$
	$implements(C', m\_C)$
	$implements(C', m\_C')$
$A - B$	$invokes(A, m\_A, B, m\_B)$
$B - C$	$invokes(B, m\_B, C, m\_C)$
$B' - B$	$isa(B', B)$
$B'' - B$	$isa(B'', B)$
$B'' - C'$	$invokes(B'', m\_B, C', m\_C')$
$C' - C$	$isa(C', C)$
$A - B$	$invokes(A, m\_A, B, m\_B) \wedge implements(B, m\_B) \wedge isa^*(B, B) \Longrightarrow icw(A, m\_A, B, m\_B)$
$B - C$	$invokes(B, m\_B, C, m\_C) \wedge implements(C, m\_C) \wedge isa^*(C, C) \Longrightarrow icw(B, m\_B, C, m\_C)$
$B'' - C'$	$invokes(B'', m\_B, C', m\_C') \wedge implements(C', m\_C') \wedge isa^*(C', C') \Longrightarrow icw(B'', m\_B, C', m\_C')$
$A - B'$	no coupling
$A - B''$	$invokes(A, m\_A, B, m\_B) \wedge isa^*(B'', B) \wedge implements(B'', m\_B) \Longrightarrow icw(A, m\_A, B'', m\_B)$
$B - C'$	$invokes(B, m\_B, C, m\_C) \wedge isa^*(C', C) \wedge implements(C', m\_C) \Longrightarrow icw(B, m\_B, C', m\_C)$
$B' - C$	no coupling
$B' - C'$	no coupling
$B'' - C$	no coupling
$A - C$	$icw(A, m\_A, B, m\_B) \wedge icw(B, m\_B, C, m\_C) \Longrightarrow icw(A, m\_A, C, m\_C)$
$A - C'$	1. $icw(A, m\_A, B'', m\_B) \wedge icw(B'', m\_B, C', m\_C) \Longrightarrow icw(A, m\_A, C', m\_C')$
	2. $icw(A, m\_A, B, m\_B) \wedge icw(B, m\_B, C', m\_C) \Longrightarrow icw(A, m\_A, C', m\_C)$

Table 1: Coupling predicates

We do not discuss each entry in Table 1 but restrict our attention to the most interesting one, the derived coupling between A and C'. It reveals that A is interaction coupled with C' via method m_A following different invocation paths and thus invoking different methods on C'. On one hand, A is interaction coupled with C' via B and B''. This case occurs when m_A invokes m_B on a member object of B''. As a consequence, the implementation of m_B at class B'' gets executed which invokes the method m_C' of class C' in turn. On the other hand, A is interaction coupled with C' via B and C. That case occurs when m_A invokes m_B on an instance of B which in turn invokes m_C on a member object of C'. Since m_C is redefined at C' the implementation of m_C given at class C' gets executed. The lessons learned from this example are twofold. Firstly, the transitive closure of interaction coupling and inheritance coupling together leads to a complex graph where it is not obvious at first sight which object classes are interaction

coupled with each other. Secondly, due to overriding and polymorphic variables it may occur that there exist several different invocation paths between two methods. Thus the same two objects of two classes may exhibit different run-time behavior at different times although initially the same method is invoked.

Summarizing, it is very important to conduct a global analysis of the classes of an object-oriented system in terms of coupling properties whenever possible since it may reveal hidden couplings which may cause problems when maintaining or extending the system at hand.

4 Cohesion

Cohesion has been defined in the realm of procedure-oriented systems [53] as "the degree of connectivity among the elements of a single module". Cohesion has been recognized as one of the most important software quality criteria. Modules with strong cohesion, in particular with functional cohesion, are easier to maintain, and furthermore, they greatly improve the possibility for reuse. A module has strong cohesion if it represents exactly one task of the problem domain, and all its elements contribute to this single task. Elements of a module are statements, subfunctions, and possibly other modules. We recall that the object-oriented counterparts of a module are methods and classes. The elements of a method are statements, local variables, and also instance variables since they are accessed either directly or via access functions in the methods. Next to methods also object classes have to be analyzed. The elements of an object class are methods and instance variables. Thus we have to distinguish the cohesion of a method from the cohesion of an object class. For the latter, we further distinguish the cohesiveness between elements directly defined within the same class from the cohesiveness between inherited and directly defined elements. Thus the following kinds of cohesion may be defined for object-oriented systems:

- method cohesion
- class cohesion
- inheritance cohesion

In the sequel we study each of the cohesion relationships in turn.

4.1 Method Cohesion

What has been stated in the realm of coupling also holds true for cohesion. Since methods equal modules to a very high degree - both bracket pieces of code implementing some functionality - we adopt the various degrees of classical cohesion [53, 59] to describe method cohesion. In contrast to coupling we do not even have to change the various notions of classical cohesion considerably. In the following, the seven degrees of classical cohesion adapted for method cohesion are summarized from worst to best. For a detailed discussion the interested reader is referred to the original paper [53].

1. **coincidental**

   The elements of a method have nothing in common besides being within the same method.

2. **logical**

   The elements with similar functionality, such as input/output handling and error handling, are collected in one method.

3. **temporal**

   The elements of a method have logical cohesion and are performed at the same time.

4. **procedural**

   The elements of a method are connected by some control flow.

5. **communicational**

   The elements of a method are connected by some control flow and operate on the same set of data.

6. **sequential**

   The elements of a method have communicational cohesion and are connected by a sequential control flow.

7. **functional**

   The elements of a method have sequential cohesion, and all elements contribute to a single task of the problem domain. Functional cohesion is the best form of method cohesion since it fully supports the principle of locality and thus minimizes maintenance efforts.

For the discussion of class cohesion and inheritance cohesion we assume that all methods have functional cohesion. The reason is, that in order to determine class/inheritance cohesion we have to investigate the relationship between methods and instance variables. Low cohesive methods which access most of the instance variables could fake a high degree of class/inheritance cohesion.

4.2 Class Cohesion

Class cohesion describes the binding of the elements defined within the same object class, not considering inherited instance variables and inherited methods. Since ignoring inheritance an object class resembles an abstract data type, and since the cohesion of abstract data types has been analyzed in detail by Embley and Woodfield in [14] we build our classification of various degrees of class cohesion on that of [14] and redefine their definitions according to the idiosyncracy of object-oriented systems. Abstract data types in procedure-oriented systems provide functionality to other abstract data types or to modules which are not abstract data types. In contrast, code in object-oriented systems is in general a method bound to a class. Thus for procedure-oriented systems with abstract

data types we have to argue which functionality we factor out to abstract data types whereas in object-oriented systems we have to consider which methods are assigned to which classes. A further crucial difference between abstract data types in the notion of Embley and Woodfield and classes is implied by the concept of object identity. Whereas a single abstract data type can export different domains an object class describes exactly one set of objects where each object is uniquely identified by some system-defined object identifier. Depending on the cohesiveness of a class its objects represent a single, semantic meaningful data abstraction or several, more or less related data abstractions. In the following we discuss the various degrees of class cohesion from worst, i.e., lowest, to best, i.e., highest.

1. **separable**

   The cohesion of a class is rated separable if its objects represent multiple unrelated data abstractions combined in one object. This is often the case if the instance variables and methods of a class can be partitioned into two or more sets such that no method of one set uses instance variables or invokes methods of a different set. In particular, the cohesion of an object class is rated separable if there is a method which does neither access any instance variable nor invokes any method of the class, or there is an instance variable which is not referenced by any of the class' methods. A class with separable cohesion should be split into several classes each representing a single data abstraction, i.e., a single semantic concept.

   Example: Consider the object class EMPLOYEE as defined above with the following extension:

   ```
   class EMPLOYEE {
       ...
       int computeCompanyRevenue (SET⟨PROJECT*⟩* p);
       ...
   };
   ```

   The method computeCompanyRevenue takes *all projects of a company* as input parameter and computes the accumulated revenue of that company. It neither accesses any instance variable of EMPLOYEE nor does it invoke any other method of EMPLOYEE. Thus the cohesion of EMPLOYEE is of separable strength. To improve its cohesion the method computeCompanyRevenue should be factored out into a different object class, e.g., into class COMPANY. □

   Note: Using syntactical criteria for partitioning an object class into disjunctive sets of instance variables and methods to detect separable cohesion is a useful aid but not applicable in some cases. For example, if the definition of an object class consists solely of n instance variables and their access methods the object class might syntactically be partitioned into n disjunctive subsets although it represents a single semantic concept. Contrarily, if a method print is defined that prints the values of all instance variables, pure syntactical analysis will classify the cohesion of the class as not separable, although it might combine one or more unrelated data abstractions.

2. **multifaceted**

The cohesion of a class is rated multifaceted if its objects represent multiple related data abstractions, accessed at least by one method. Separable cohesion can often be detected by a syntactical analysis of the class definition. For multifaceted cohesion, however, we must always look at the semantics of a class and its elements. Similar to separable cohesive classes a multifaceted class covers different semantic concepts. Yet, at least one method references instance variables or invokes methods of the different semantic concepts, such that the cohesion of the corresponding class cannot be rated separable. A semantic analysis is necessary for determining multifaceted cohesion. Well-known data modeling concepts like well-formed entity-relationship modeling [35], and data normalization [1, 12, 20] may be used for that purpose. To be able to apply data normalization theory we adapt the definition of multifaceted cohesion as follows. The cohesion of a class is rated multifaceted if the set of instance variables of the class interpreted as relation schema is not in second normal form. To put it in other words, the instance variables describe two ore more semantic concepts. Thus, they are only dependent on part of the user-defined key. However, they are not separable since a method defined on several instance variables exists. A multifaceted class should either be split into several classes each of which representing exactly one semantic concept with the set of instance variables being at least in second normal form, or some instance variable(s) should be moved to a class referenced by another instance variable. The method which leads to multifaceted cohesion should be assigned to and possibly reimplemented in one of the newly created classes. It depends on the problem domain to which class the method should be assigned.

Example: Consider object class REORDER:

```
class REORDER {
      ITEM* reorderedItem;
      COMPANY* reorderedFrom;
      int discount;
      int quantity;
      ...
public:
      bool expectedRevenue ();
      ...
};
```

The method expectedRevenue computes the revenue expected by determining the difference between the price of an item and the discount given by the company and by multiplying this difference with the quantity of the reordered item.

If we interpret the set of instance variables as attributes of a relation schema attributes recordedItem and recorderedFrom form the key of this relation schema. Assume, the discount given depends only on the company, i.e., it is the same for all items. Thus, the second normal form is violated. This definition of REORDER has multifaceted cohesion. To improve its cohesion the instance variable discount should be moved to class COMPANY:

```
class COMPANY {
      STRING* name;
      real discount;
      ...
public:
      real discount ();
      ...
};
```

□

3. **non-delegated**

The cohesion of a class is rated non-delegated if it is neither separable nor multifaceted and if one method uses instance variables which describe only a component of the respective class. Hence, we may again use data normalization theory to detect non-delegated cohesion like we did for the analysis of multifaceted cohesion. For this purpose, we also adapt the definition of non-delegated cohesion as follows. The cohesion of an object class is rated non-delegated if the set of instance variables interpreted as relation schema is not in third normal form. To put it in other words, there exist instance variables which do not describe the whole data abstraction represented by the class but only a component of it. To overcome non-delegated cohesion the "non delegated" methods and instance variables should be delegated to the component classes on which they are actually defined.

Example: Consider again the object class EMPLOYEE:

```
class EMPLOYEE {
      STRING* name;
      DATE* birthDate;
      PROJECT* involvedInProject;
      EMPLOYEE* managerOfProject;
      ...
public:
      float computeSalary ();
      bool managerIncomeHigherThanAverageInProject ();
      ...
};
```

If we interpret the set of instance variables as attributes of a relation schema the attribute name is the key of this relation schema. The attributes birthDate and involvedInProject depend directly on the attribute name. However, the attribute managerOfProject depends directly on the project referenced by involvedInProject and transitively on name, thus the third normal form is violated. This definition of EMPLOYEE has non-delegated cohesion. To improve its cohesion the instance variable managerOfProject and the method managerIncomeHigherThanAverageInProject should be delegated to the object class PROJECT:

```
class PROJECT {
      EMPLOYEE* managerOfProject;
```

```
        SET⟨EMPLOYEE*⟩* membersOfProject;
        DATE* startDate;
        DATE* expectedEndDate;
        ...
public:
        bool managerIncomeHigherThanAverageInProject ();
        SET⟨EMPLOYEE*⟩* getProjectMembers ();
        ...
};
```

□

Unfortunately, it is not always as obvious as in the example above where to place methods. The placement of methods also raises several questions concerning the visibility and the possible invocation of methods. If a method is delegated to a component class because it mainly uses instance variables of that component class we have to consider if and how this method is visible to clients of the inspected class, i.e., the class under consideration. One way to organize the invocation of the delegated method is that clients of the inspected class receive a handle to the component object, thus, they are able to directly invoke the method on the component object. This solution might not always be desirable since it means that the inspected class exhibits parts of its implementation, i.e., its components. Furthermore, this solution may increase the number of classes the clients of the inspected class have to interact with. Thus the component coupling between a client class and the component class may be made worse.

Another possibility to organize the invocation of the components' methods is to hide the component structure completely from the client classes and to let them access the methods of the component classes through so called propagation methods of the inspected class. This solution is recommended by the *Law of Demeter* from Lieberherr et al. [30, 31, 32, 33]. The law states that a method m of some class C may only invoke methods of such classes which are used as domains of instance variables of C, or as domains of input parameters of m. Objects which are newly created within m may also be the receivers of messages. Essentially, the law prohibits method invocation on objects which have been returned by some other method. One goal of the Law of Demeter is to decrease the coupling between different object classes by restricting the object classes with which a specific class may communicate. However, the decrease of component coupling is traded for an increase of interaction coupling between the client classes and the inspected class and between the inspected class and the component class, respectively. For each method of the component class which should be visible to the client class a propagation method has to be implemented at the inspected class. The client class calls the propagation method, which calls the corresponding method in the component class in turn. Besides the increase of interaction coupling, propagation methods also introduce *tramp data*, i.e., data which is passed from the client class via the inspected class to the component class without being used in the inspected class at all. Tramp data increases interaction coupling between the client class and the inspected class

since a change of the definition of this data implies a change of the inspected class although the data is not used.

At first sight there exists a contradiction between the Law of Demeter and the goal to avoid non-delegated cohesion. Whereas the former introduces propagation methods, i.e., non-delegated methods, to reduce component coupling, the latter factors out non-delegated methods to component classes to increase class cohesion and – as side-effect – to reduce interaction coupling. A solution to overcome this dilemma is to consider the situation from a semantic point of view. If the component structure describes a relationship between conceptually different objects, as revealed by semantic data modeling, it is favorable to reveal the component hierarchy in order to avoid propagation methods with tramp data. But, if the component structure is mainly an implementation detail, the solution with propagation methods is more appropriate.

4. **concealed**

   The cohesion of a class is rated concealed if it neither has separable, nor multifaceted, nor non-delegated cohesion, but there exists some useful data abstraction concealed in the data abstraction represented by the class. In modules, concealed cohesion resembles a piece of inline code which can be factored out to a subroutine or function. In analogy, a class with concealed cohesion includes some instances variables and referencing methods which may be regarded as a class of its own. Factoring out of such parts has two advantages. Firstly, the structure of the inspected class becomes more intuitive and more concise, thus increasing the class' cohesion. Secondly, it permits the new class being reused as component in other classes as well.

   Factoring out such a concealed class implies that the instance variables factored out are replaced by one instance variable referencing an object of the new class. All methods of the inspected class referencing only factored out instance variables are factored out, too. For some methods, one has to decide whether to place them in the new class or to leave them in the inspected class according to the discussion above about non-delegated cohesion.

   Example: Consider again the object class EMPLOYEE with the instance variables name, jobProfile, dayOfBirth, monthOfBirth, and yearOfBirth, and dayOfHire, monthOfHire and yearOfHire. The instance variables describing various dates may be factored out to a new class DATE with the instance variables day, month, and year. The respective instance variables of the class EMPLOYEE are then replaced by two instance variables birthDate and hireDate. □

   Candidates for new classes are on one hand instance variables with complex domains, and on the other hand sets of instance variables which are often used together in methods but rarely used together with other instance variables. Some kind of cluster analysis may exhibit candidates. When creating a new class we have to keep in mind that the class should have the best cohesion characteristic possible.

5. **model**

Model cohesion is the highest degree in our classification. The cohesion of a class is rated model if the class represents a single, semantically meaningful concept without containing methods which should be delegated to other classes and without containing concealed classes.

It is interesting to note that a semantically similar notion, *informational strength*, has been already defined by Myers [44] in the realm of module-oriented systems. There, a module is interpreted as implementation of an abstract data type. Taking the classical function-oriented notion of cohesion a bit further Myers defines a module to have informational strength if all its functions have functional cohesion and they work on the same set of data.

4.3 Inheritance Cohesion

Whereas class cohesion only inspects the binding of the newly defined elements within a class, inheritance cohesion also takes the inheritance hierarchy into account. It describes the binding of the newly defined elements together with the inherited elements. Since the latter are transitively inherited from direct and indirect superclasses inheritance cohesion evaluates not only the cohesion of an immediate class-superclass relationship but inspects the whole inheritance hierarchy. Inheritance cohesion is strong if this hierarchy is a generalization hierarchy in the sense of conceptual modeling, and it is weak if the inheritance hierarchy is merely used for code sharing among otherwise unrelated classes.

Since the aim for each newly defined subclass is to exhibit a single semantic concept we may use the same classification for inheritance cohesion as it was defined for class cohesion.

5 Outlook

In this paper we introduced a comprehensive taxonomy of coupling and cohesion characteristics of object-oriented systems. In contrast to the classical notion of coupling and cohesion being based on a single concept, the module, there are two subjects of interest, methods and object classes, in the realm of object-oriented systems. In addition, the important concept of inheritance considerably influencing the structure of an object-oriented system has to be taken into account. This leads to three interrelated dimensions of coupling and cohesion, respectively: interaction coupling, component coupling, and inheritance coupling, as well as method cohesion, class cohesion, and inheritance cohesion.

The goal of the paper has been to define qualitative criteria for coupling and cohesion to improve the quality of object-oriented systems. The paper deliberately did not aim at metrics and quantitative criteria. The reason is not at all that these aspects are not important. However, it is our firm belief that at the current state of art of object-oriented quality assessment it is very important to uncover the various aspects of quality criteria. Once these criteria are commonly accepted, metrics and automatic quality assessors may be developed. Furthermore, the paper deals only with coupling and cohesion properties. Software quality, however, does not depend exclusively on these two properties but on

various other factors like method fan-in/fan-out, and reuse via inheritance and polymorphism, to mention just a few. To reach a comprehensive assessment of software quality all factors have to be considered. In case of conflicting goals individual, problem dependent decisions have to be taken. Finally, we did not take any performance considerations into account when defining the various degrees of coupling and cohesion. The reason is that coupling and cohesion are criteria for evaluating object-oriented designs in the first place. The designs may be tuned during implementation for performance reasons.

Further research expands mainly into three areas. Firstly, a comprehensive analysis of existing object-oriented design guidelines concerning their influence on coupling and cohesion quality will help in unifying and consolidating the various approaches to good designs. Secondly, as soon as a common understanding of coupling and cohesion characteristics is reached tools for (partially) automatic assessment of coupling and cohesion properties will be developed. There is some confidence that these tools not only aid in designing good object-oriented software but are a useful means when searching class libraries and identifying related object classes [36]. Lastly, the long-term goal comprises the investigation of other quality criteria of object-oriented systems and their interdependencies with the coupling and cohesion characteristics defined in this paper.

Acknowledgement

The authors are grateful to Dieter Merkl, Roland Mittermeir, David Monarchi, Georg Reichwein, and Markku Sakkinen for valuable comments on earlier versions of this paper.

References

[1] E. Andonoff, "Normalization of Object-Oriented Conceptual Schemes," in *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE'93)* , ed. C. Rolland, F. Bodart and C. Cauvet, pp. 449-462, Springer LNCS 685, 1993.

[2] S.C. Bailin, "An automated quality assessor for Ada object-oriented designs," in *Proc. of the 40th IEEE National Aerospace and Electronics Conference*, vol. 2, pp. 732-738, Piscataway, 1988.

[3] E. V. Berard, *Essays on Object-Oriented Software Engineering, Vol.I*, Prentice-Hall, 1993.

[4] B.W. Boehm , J.R. Brown and M. Lipow , "Quantitative Evaluation of Software Quality," in *IEEE 2nd International Conference on Software Engineering*, pp. 286-299, 1976.

[5] G. Booch, *Object-Oriented Analysis and Design with Applications (2nd edition)*, Benjamin Cummings, 1994.

[6] T.A. Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1991.

[7] L. Cardelli, "A Semantics of Multiple Inheritance," in *Journal of Information and Computation*, vol. 76, pp. 138-164, 1988.

[8] L. Cardelli et al., "Modula-3 Language Definition," in *ACM SIGPLAN Notices*, vol. 27, pp. 15-42, August 1992.

[9] S.R. Chidamber and C.F. Kemerer, "Towards a Metric Suite for Object-Oriented Design," in *Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices*, vol. 26, Oct. 1991.

[10] S.R. Chidamber and C.F. Kemerer, "A Metric Suite for Object-Oriented Design," in *IEEE Transactions on Software Engineering*, vol. 20, pp. 476-493, June 1994.

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

[12] C.J. Date, *Relational Database: Selected Writings*, Addison Wesley, 1986.

[13] M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[14] D.W. Embley and S.N. Woodfield, "Cohesion and Coupling for Abstract Data Types," in *6th International Phoenix Conference on Computers and Communications*, pp. 292-234, IEEE Computer Society Press, Arizona, 1987.

[15] D.W. Embley and S.N. Woodfield, "Assessing the Quality of Abstract Data Types Written in Ada," in *International Conference on Software Engineering*, pp. 144-153, IEEE Computer Society Press, 1988.

[16] G. Engels and G. Kappel, "Object-Oriented Systems Development: Will the New Approach Solve Old Problems?," in *Proceedings of the IFIP94 World Congress, Vol.III*, ed. K. Duncan and K. Krueger, North-Holland, 1994.

[17] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, 1984.

[18] B. Henderson-Sellers, S.C. Bilow and W. Harrison, "Workshop Report on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics at OOPSLA'94," in *OOPS Messenger*, vol. 5, pp. 78-80, October 1994.

[19] M. Hitz and B. Montazeri, "Measuring Product Attributes of Object-Oriented Systems," in *European Software Engineering Conference (ESEC'95)*, Sept. 1995 (to appear).

[20] S. Hong, "A Class Normalization Approach to the Design of Object-Oriented Databases," in *Proc TOOLS USA '91 (Technology of Object-Oriented Languages and Systems)*, pp. 63-71, 1991.

[21] Hood Working Group, *The HOOD (Hierarchical Object-Oriented Design) User Manual, Issue 3.0*, European Space Agency, Sept. 1989.

[22] C. Jean and A. Strohmeier, "An Experience in Teaching OOD for Ada Software," in *ACM SIGSOFT Software Engineering Notes*, vol. 15, pp. 44-49, Oct. 1990.

[23] R.E. Johnson and B. Foote, "Designing reusable classes," in *Journal of Object-Oriented Programming (JOOP)*, vol. 1, pp. 22-35, 1988.

[24] R. Jungclaus, G. Saake, T. Hartmann and C. Sernadas, "TROLL - A Language for Object-Oriented Specification of Information Systems," ACM Transactions on Information Systems (to appear), 1995.

[25] G. Kappel and M. Schrefl, "Object/Behavior Diagrams," in *Proceedings of the 7th International Conference on Data Engineering*, pp. 530-539, IEEE Computer Society Press, Kobe, Japan, April 1991.

[26] G. Kappel, J. Overbeck and M. Schrefl, "A Process Model for Object-Oriented Development," Technical Report MOOD 92/08, presented at *Workshop on Object-Oriented Software Development Process at ECOOP'92*, Utrecht, July 1992.

[27] S. Keene, *Object-Oriented Programming In Common Lisp: A Programmers Guide To Common Lisp Object System*, Addison-Wesley, 1988.

[28] T. Korson and J.D. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," in *Communications of the ACM*, vol. 33, pp. 40-60, 1990.

[29] O. Lehrmann-Madsen, B. Møller-Pedersen and K. Nygaard, *Object-Oriented Programming in the BETA Programming Language*, Addison Wesley, 1993.

[30] K.J. Lieberherr, I. Holland and A.J. Riel, "Object-Oriented Programming: An Objective Sense of Style," in *Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices*, ed. N. Meyrowitz, vol. 23, pp. 323-343, Nov. 1988.

[31] K.J. Lieberherr and I. Holland, "Assuring good style for object-oriented programming," in *IEEE Software*, pp. 38-48, Sept. 1989.

[32] K.J. Lieberherr and C. Xiao, "Formal Foundations for Object-Oriented Data Modeling," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, pp. 462-478, June 1993.

[33] K.J. Lieberherr, I. Silva-Lepe and C. Xiao, "Adaptive Object-Oriented Programming using Graph-Based Customization," in *Communications of the ACM (CACM)*, vol. 37, pp. 94-101, May 1994.

[34] B. Liskov and J.M. Wing, "A New Definition of the Subtype Relation," in *Proc of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, ed. O.M. Nierstrasz, pp. 118-141, Springer LNCS 707, Kaiserslautern, 1993.

[35] V.M. Markowitz and A. Shoshani, "On the Correctness of Representing Extended ER Structures in the Relational Model," in *Proceedings of the ACM-SIGMOD Int. Conf. on the Management of Data*, pp. 430-439, Portland, 1989.

[36] D. Merkl, A M. Tjoa and G. Kappel, "Structuring a Library of Reusable Software Components Using An Artificial Neural Network," in *Proc of the 2nd International Conference on Achieving Quality in Software (AQuIS'93)*, Venice, Oct. 1993.

[37] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[38] B. Meyer, "Tools for the New Culture: Lessons from the Design of the Eiffel Class Library," in *Communications of the ACM*, vol. 33, pp. 68-88, 1990.

[39] B. Meyer, *Eiffel: The Language*, Prentice-Hall, 1992.

[40] R.T. Mittermeir, "Design Aspects Supporting Software Reuse," in *Software Reuse*, ed. L. Dusink and P. Hall, pp. 115-119, Springer, 1989.

[41] D.R. Moreau and W.D. Dominick, "Object-Oriented Graphical Information Systems: Research Plan and Evaluation Metrics," in *The Journal of Systems and Software*, vol. 10, pp. 23-28, 1989.

[42] D.R. Moreau and W.D. Dominick, "A Programming Environment Evaluation Methodology for Object-Oriented Systems: Part I - The Methodology," in *Journal of Object-Oriented Programming (JOOP)*, pp. 38-52, May/June 1990.

[43] D.R. Moreau and W.D. Dominick, "A Programming Environment Evaluation Methodology for Object-Oriented Systems: Part II - Test Case Applications," in *Journal of Object-Oriented Programming (JOOP)*, pp. 23-32, September/Oct. 1990.

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

[45] K.W. Nielsen, "Task coupling and cohesion in Ada," in *Ada Letters*, vol. 6, pp. 44-52, July/August 1986.

[46] O.M. Nierstrasz, "A Survey of Object-Oriented Concepts," in *Object-Oriented Concepts, Databases and Applications*, ed. W. Kim and F. Lochovsky, pp. 3-21, ACM Press and Addison-Wesley, 1989.

[47] O. Nierstrasz, S. Gibbs and D. Tsichritzis, "Component-Oriented Software Development," in *Communications of the ACM* , vol. 35, pp. 160-165, 1992.

[48] M. Page-Jones, *The Practical Guide to Structured Systems Development*, Yourdon Press, 1980.

[49] M. Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence," in *Communications of the ACM*, vol. 35, pp. 147-151, Sept. 1992.

[50] D. Rocacher, "Smalltalk-80 and Smack: towards a methodological approach to software quality," in *Proc. Esprit'88 Putting the Technology to Use*, pp. 492-507, North Holland, 1988.

[51] C. Schaffert et al., "An Introduction to Trellis/Owl," in *Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices*, ed. N. Meyrowitz , vol. 21, pp. 9-16, Dec. 1986.

[52] A. Snyder, "Inheritance and the Development of Encapsulated Software Components," in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner, pp. 165-188, The MIT Press, 1987.

[53] W. Stevens, G. Myers and L. Constantine, "Structured Design," in *IBM Systems Journal*, vol. 13, pp. 115-139, 1974.

[54] J. Stiebellehner, G. Kappel and H. Schauer, "Kopplungs- und Konzentrationszahlen für objektorientierte Software," in *Bericht des GI-Workshops "Software-Metriken"*, Magdeburg (BRD), Sept. 1994.

[55] D.P. Tegarden, S.D. Sheetz and D.E. Monarchi, "A Software Complexity Model of Object-Oriented Systems," in *Journal of Decision Support Systems*, 1994.

[56] P. Wegner, "Dimensions of Object-Based Language Design," in *Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices*, vol. 22, pp. 168-182, Dez. 1987.

[57] A. Wirfs-Brock and B. Wilkerson, "Variables Limit Reusability," in *Journal of Object-Oriented Programming (JOOP)*, pp. 34-40, May/June 1989.

[58] R. Wirfs-Brock, B. Wilkerson and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

[59] E. Yourdon and L.L. Constantine, *Structured Design*, Prentice-Hall, 1979.