An Information-Based View of Representational Coupling in Object-Oriented Systems

Pierre Kelsen*

Luxembourg University of Applied Sciences, Department of Applied Computer Science, L-1359 Luxembourg-Kirchberg

Abstract. In this paper we investigate a special type of coupling in object-oriented systems. When a method of a class C invokes a method of a class D, the method of C becomes dependent on the representational details of D: the more low-level the service provided by D is the higher the dependency of C on D. This dependency is known as *representational coupling*. Coupling in general, and representational coupling in particular, are important because they influence the extensibility of a system, that is, the ease with which software can be adapted to changing requirements: the higher the coupling the harder it is to make changes since any changes local to one module are likely to affect many other modules.

We propose a *qualitative measure* of representational coupling (as opposed to quantitative measures provided by metrics) that is based on partial orders over equivalence relations on the state space. We also introduce the notion of *intrinsic representational coupling* that expresses the amount of representational coupling that is inherent to the system. Finally, we show that despite its non-quantitative nature our measure can be useful in identifying candidate methods for refactoring. We demonstrate this by applying our measure to several examples in the literature, showing in each case how an implementation with non-minimal representational coupling can be transformed using a few simple refactorings into a solution with minimal representational coupling (equal to the intrinsic representational coupling).

Keywords. coupling, object-oriented, extensibility, refactoring, metrics

1 Introduction

Coupling is a well-known quality factor of software. It expresses how strongly individual software modules depend on each other. Coupling has a major impact on the extensibility of software, that is, on the ease with which software can be modified to adapt to changing requirements. The lower the coupling the easier it is to change individual modules since changes local to one module are likely

* Work supported by the Luxembourg Ministry of Culture, Higher Education and Research under grant IST/02/03

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 216–230, 2003.

to have a lower impact on other modules. There exists a similar relationship between coupling and fault-proneness, that is, the likelihood of detecting a fault in a class: the higher the coupling in a system, the higher the fault-proneness. The relationship between coupling and external quality attributes such as fault-proness has been demonstrated by defining metrics for various types of coupling and performing empirical studies on the relevance of these metrics (e.g., [BBM96, Bri97,LH93]).

In this paper we consider representational coupling, a fundamental type of coupling ubiquitous in object-oriented systems. Whenever an object calls a method of another object, this method call reveals something about the callee: in the worst case (resulting in high representational coupling) it discloses implementation level information, in the best case (low representational coupling) it provides high-level information that reveals little about the internal structure of the object. The term representational coupling was first used in [Cha93] and discussed in some detail in [Rich99].

For example if the invoked method is simply the accessor function for an attribute, we have a high degree of representational coupling since these accessor functions are close to the implementation level (and thus likely to change). On the other hand if the method that is invoked provides a much higher-level service, then implementation changes in that method‘s class are less likely to affect this service, thus resulting in a lower degree of representational coupling.

As we shall explain later in this paper existing coupling metrics (such as [ChiKem91,ChiKem94,LH93,Lee95,Bri97,Yac99,Ari02]) do not capture representational coupling. One reason for this is that they are based on counting different types of interactions. The resulting quantitative measures are too coarse-grained to evaluate the level of abstraction of method calls, something that is needed for evaluating representational coupling.

In this paper we develop a "qualitative measure" for representing representational coupling: rather than simply assigning an ordinal number to a given design, we base our measure on the refinement ordering over equivalence relations on the state space. We also develop the notion of intrinsic representational coupling, which is a measure for the minimum amount of representational coupling inherently contained in a system. We shall prove that no design can have representational coupling less than the intrinsic representational coupling.

Because of the qualitative nature of our measure it cannot be directly used as an indicator to make predictions on external quality factors (such as fault-proneness or extensibility). It can however be used to compare the representational coupling with the minimum representational coupling possible as given by the intrinsic representational coupling. In this manner our measure allows us to detect possible candidates for refactoring. We shall exemplify this application by considering three examples from the literature. For each of these examples we describe an ad-hoc solution that displays non-optimal representational coupling. By using a few simple refactoring techniques we are able to transform each of these implementations into a solution with representational coupling equal to the intrinsic representational coupling.

In Sect. 2 we introduce basic object-oriented terminology. In Sect. 3 we derive a mathematically precise definition of representational coupling. In Sect. 4 we define the notion of intrinsic representational coupling. In Sect. 5 we show by means of examples how we can achieve optimal coupling using simple refactorings. In the final section we present some conclusions of our work.

2 Object-Oriented Terminology

An object-oriented system is made up of *objects* which are instances of *classes*. Each class consists of *methods* and *attributes*. We consider only non-static methods. The allowed data types of attributes depend on the programming language. We shall simplify the following definitions by basing ourselves on Java. These definitions can easily be adapted to other languages (such as C++, Eiffel or Smalltalk).

Since we attempt in this paper to establish a sound mathematical basis for representational coupling, we need to exercise some care in defining the basic terms unambiguously. An *object* is really a triplet $(identifier, type, value)$ where the identifier is from a set of identifiers I and the value - which we also call the *state of the object* - is a tuple $[A_1 : v_1, \ldots, A_k : v_k]$ where each v_i is the value of attribute named A_i. The value of an attribute is defined as follows: if the attribute is of a primitive type then v_i is the primitive value, if it is of a class type then the value is either null or an identifier in I, and finally, if it is of an array type, then its value is the tuple of values of the individual elements (recursive definition!). An *object set* consists of a set of objects such that different objects have distinct identifiers in I and each identifier of I occurs as identifier of an object in the set.

We refer to the pair (identifier set, object set) as the *system state*. The system starts out in an *initial state*. Any system state that can be reached from this initial state is called a *reachable state*. From here on whenever we talk about a state we imply that the state must be reachable.

3 Towards a Formal Definition of Representational Coupling

3.1 Related Work on Coupling

Much research effort has been spent on coupling in the field of software metrics. Several types of coupling in object-oriented systems have been investigated. In [Bri99a] three frameworks ([Eder94,HiMo95,Bri97]) for describing coupling are investigated and an attempt is made to unify those frameworks according to certain criteria. One of these criteria is the type of connection, that is, the mechanisms that may contribute to coupling. The following classification of connections is given (assume m is a method of class C and D denotes another class):

1. D is the type of an attribute of C
2. D is the type of a parameter or a return type of m
3. D is the type of a local variable of m
4. D is the type of a parameter of a method invoked by m
5. m references an attribute of D
6. m invokes a method of D
7. high-level dependency of C on D such as "uses"

Representational coupling corresponds to connection type 6. Connection types 1-4 involve class D as a type being used in C. Connection type 7 only measures coupling at a very high level. Of the two remaining types connection type 6 is in fact the only type of coupling that takes into account the methods of the server class. In [Bri99a] a list of metrics developed for each connection type is also given: from that list one can see that the largest number of metrics was indeed developed for connection type 6, thus underscoring its central role in coupling measurement.

Let us take a closer look at those metrics:

1. Coupling between Objects ([ChiKem91,ChiKem94]): counts the number of other classes to which a class is coupled.
2. Response for class ([ChiKem91,ChiKem94]): the number of methods that can be potentially executed in response to a message received by an object of that class.
3. Message passing coupling ([LH93]): number of send statements in a class.
4. Information-flow-based coupling([Lee95]): counts for a method of a class the number of polymorphically invoked methods, weighed by the number of parameters of the invoked method.
5. In [Bri97] a suite of measures is proposed that count various types of interactions (e.g., class-attribute, class-method, method-method,...).
6. More recently dynamic (runtime) metrics where developed ([Yac99,Ari02]): these are based on counting the number of messages sent (or received) by an object during runtime.

Without going into the details of the definitions of these measures we observe that they are all based on counting certain types of interactions. They are do not permit a deeper analysis of the interaction between a method and a specific class, something that is needed to "measure" the abstraction level of an interface used by a method.

3.2 Example: An Elevator Control System

En route to a more rigorous approach to representational coupling a concrete example will be of great help. The following example, taken from [Rich99] describes an elevator control system. The system contains two classes: ElevatorControl and Elevator. The class Elevator represents an elevator in a building. Each elevator has a direction and a position, yielding two methods direction() and position() in the Elevator class. The ElevatorControl class has a single responsibility, namely

handling requests. Thus, if somebody pushes a button on a certain floor this class has to dispatch an elevator to take care of that request. Accordingly ElevatorControl maintains a list of elevators and has a single method, handleRequest() that has as argument an object of class Request.

In the first implementation the *handleRequest()* function polls each elevator, asking for its direction and position. Based on this information the method computes the closest elevator and assigns the request to that elevator. Note that the information the ElevatorControl class asks from the Elevator class, namely direction and position, is rather low-level.

Here is the pseudo-code describing the first implementation:

Implementation 1 *Method ElevatorControl.handleRequest(Request r)*

```
1. float minDist = infty;// positive infinity
2. Elevator ec = null; // reference to closest elevator
2. for each elevator e do {
3.    compute d=distance(e,r) using e.direction() and e.position();
4.    if (d< minDist) {
5.       ec = e;
6.       minDist = d;
7.    }
8. }
9. if (ec!= null)
10. ec.addRequest(r);
```

In a second implementation the Elevator class offers a higher-level interface: it provides a single public function *computeDistance(Request r)* which computes the distance of this elevator to the request. In this implementation the handleRequest method simply asks each elevator to compute its distance from the request and then assigns, as before, the request to the closest elevator. All that changes in the implementation is that the pseudo-code on line 3 above is replaced by

```
d=e.computeDistance(r);
```

Intuitively in the second implementation the handleRequest method has lower representational coupling with respect to the Elevator class since it accesses a higher-level interface. This intuition is seconded by analyzing how changes to the Elevator class affect the system design. For instance suppose we want to take into account the fact that an elevator may be out of order. In this case we have to add some attribute to the ElevatorClass as well as a method outOfOrder(). In the first implementation we also have to adapt the implementation of the handleRequest function. In the second implementation it suffices to let the computeDistance return a very large number (e.g., positive infinity) to indicate that an elevator is out of order without affecting the ElevatorControl class.

Similarly if we want to consider service elevators that move more slowly or elevators that can only stop on certain floors, both the ElevatorControl and the

Elevator class need to be changed in Implementation 1 while the changes are restricted to the Elevator class in Implementation 2. (See [Rich99] for details.)

3.3 An Information-Based View

We take the previous example as the starting point towards a more formal definition. Recall that in Implementation 1 the method handleRequest() invokes the direction() and position() methods of the Elevator class while in the second implementation handleRequest() only uses the computeDistance() method of Elevator.

Let us compare the two implementations from an information-theoretic standpoint. The reason we can replace the calls to direction() and position() by a call to computeDistance() is that method handleRequest() does not actually need all the information conveyed by the first two low-level functions. Indeed we do not need to distinguish the two cases where an elevator has a different position and direction but the same distance to the request. This observation allows us to use the computeDistance() method since it conveys just enough information to the handleRequest() method.

Since a formalization based on the abstraction level of an interface seems difficult and since the informal notion of representational coupling appears to be related to the information exchanged between a method and a class, we put our measure of representational coupling on an information-theoretic base.

The concept of *message* is central to our analysis since it is the vehicle by which information between a method and a class is exchanged.

Definition 1 *A message is a 4-tuple* $(a, b, methodName, args)$ *where* a *and* b *are object identifiers,* $methodName$ *is the name of a method of object* b *and* $args$ *denotes the argument values for that method. Note that* $args$ *may itself be a tuple if the method takes several arguments. This notation means that the message is sent from object* a *to object* b *and invokes method* $b.methodName()$ *with arguments* $args$*.*

A method call may return a result. For the sake of uniformity, we shall assume that for every message that corresponds to a method call, there is a special *return message*. Thus, the return message for a message $(a, b, methodName, args)$ is a message $(b, a, return, result)$ where *return* is a reserved name and *result* is the result value of the function, or *void* if the function does not return a value.

For the following discussion we shall assume that a method m is executed on an object of class C. We shall analyze the information exchanged between method m and a second class D by examining the sequence of messages that enter or leave an object of D. For a given implementation of m this sequence depends on

- the system state s (see Sect. 2),
- the choice of an object c of class C (= object on which method m will be executed),

– the choice of an object d of class D (at which we observe the sequence of messages).

Here we assume that c and d are identifiers in the identifier set of system state s (see Sect. 2). We call the sequence of messages obtained for a given choice of s, c and d the *actual message sequence*. We shall call the function that assigns to each triplet (s, c, d) an actual message sequence the *message sequence function*. We also say that *m interacts with class D through message sequence function q*.

To illustrate these notions consider the elevator control system. In the first implementation the actual message sequence of an Elevator object consists of four messages: the position and direction query messages together with their return messages. In the second implementation the actual message sequence comprises only two messages: the computeDistance request and the return message with the result. Note that in both cases the actual message sequence depends on the system state and on the choice of the Elevator object since for different system states the Elevator object may have a different position and/or direction possibly resulting in different return messages (having different result fields). In neither case is the message sequence function a constant function.

To evaluate the amount of information exchanged between method m and object d, we could simply take into account the total size of all messages in the actual message sequence but such a measure would be highly dependent on the particular encoding chosen. Another problem with this approach is the necessity to deal with object references: how does one measure the information contained in an object reference?

We use a different approach based on examining the dependency of the actual message sequence on the internal state of the object d (at which we observe the sequence of messages).

Definition 2 *A message sequence function q* **induces an equivalence relation** *on $S(D)$, the set of possible states of an object of class D as follows: for $s, s' \in S(D)$ we define $s \equiv_q s'$ if and only if $q(s_0, c, d) = q(s'_0, c, d)$ (ie, q yields the same actual message sequence) for any choice of c and d (of types C and D) and for any two system states s_0 and s'_0 that differ only in the state of object d where they have values s and s', respectively.*

Intuitively, two states of an object of D are equivalent if they cannot be distinguished by method m (since they produce the same actual message sequence regardless of the states of the other objects).

As noted earlier, in the elevator example the distance from the request can be computed from the position and direction of the elevator. We can express this in terms of the induced equivalence relations: the message sequence function for the first implementation induces an equivalence relation on the state space of the Elevator class that is a strict refinement of that induced by the message sequence function for the second implementation. We conclude that the elevator class reveals less information in the second implementation. (Recall that an equivalence relation is a *refinement* of another equivalence relation defined over

the same ground set if the equivalence classes of the former equivalence relation are subsets of the equivalence classes of the latter relation.)

Method m can be implemented in many different ways. In each case m interacts with class D through a possibly different message sequence function. We need to be able to compare these functions.

Definition 3 *Let q and q' be two message sequence functions. Message sequence function q is weaker than message sequence function q', written as $q \prec q'$ if q' induces an equivalence relation over $S(D)$ that is a strict refinement of the equivalence relation induced by q.*

We observe that this ordering on message sequence functions is not a partial order (since it is not anti-symmetric) but the refinement ordering on the corresponding equivalence relations is indeed a partial order.

We can apply this definition to representational coupling as follows: consider two implementations m and m' with "the same behavior" in a class C. (Two methods have the same behavior if from any possible initial state they produce the same final state.) Then we say that m has lower representational coupling than m' with respect to a class D if m interacts with D through a weaker message sequence function than m'.

4 Intrinsic Representational Coupling

Let $S_{m\langle c}$ be the function that maps any system state to another system state based on the action of m invoked on object c. In a sense $S_{m\langle c}$ describes the semantics of method m. That is, if we call $c.m()$ on initial system state s_0, then the system will be in state $S_{m\langle c}(s_0)$ after m terminates. We call the function $S_{m\langle c}$ the *state mapping* for method m. In this section we take the view that the same method may have different implementations yielding the same state mapping and thus having the same behavior. In this context it makes sense to speak of a method interacting with a class through different message sequence functions (corresponding to different implementations with the same behavior).

Suppose a method m interacts with class D through a certain message sequence function. How can we tell whether there is an implementation for m that yields an even weaker message sequence function? Or maybe even a weakest possible message sequence function. This would imply in a sense that m has some intrinsic degree of representational coupling.

In this section we give some answers to these questions. For a system state s and an object b of s, let $s - b$ denote the system state with the same object set as s, but without the state information for b (the states given for all other objects are the same as those in s).

Definition 4 *Method m induces an equivalence relation on the set $S(D)$ of possible states of d as follows: for $s, s' \in S(D)$ we set $s \equiv_m s'$ if and only if for any two system states s_1 and s'_1 that differ only on d where they have states s and s', respectively, we have, for any choice of object c (of type C), $S_{m\langle c}(s_1) - d = S_{m\langle c}(s'_1) - d$.*

Loosely expressed this means that s and s' cannot affect the state outside d in different ways.

Theorem 1. *Suppose that m interacts with class D through message sequence function q. Then the equivalence relation induced by q (see definition 2) is a (not necessarily strict) refinement of the equivalence relation induced by m.*

Proof. Assume for a contradiction that this is not the case. Then there exist two states s and s' in $S(D)$ such that $s \equiv_q s'$ but $s \not\equiv_m s'$. In other words, there exist two system states s_1 and s'_1 that differ only on d where they have values s and s', respectively, and having the following two properties: (1) they produce the same actual message sequence, and (2) they yield a different state for an object other than d. But this is not possible: if the actual message sequence is the same then the states of objects other than d must remain the same also since the system looks exactly the same in both system states for any object other than d. QED

Lemma 1. *Suppose that m interacts with D through message sequence function q and that the equivalence relation induced by q on the state space of D is equal to the equivalence relation induced by m. Then m cannot interact with D through a message sequence function that is weaker than q.*

Proof. If m interacts with D through a weaker message sequence function, then the equivalence relation induced by that message sequence function is not a refinement of the equivalence relation induced by m thus contradicting the previous theorem. QED

If a message sequence function for an implementation satisfies the condition of the lemma, then we will say that the underlying implementation has *minimal representational coupling* with respect to class D equal to the *intrinsic representational coupling* of method m with respect to class D (implying that no other implementation can yield a weaker message sequence function).

5 Reducing Representational Coupling via Refactorings

The technique for reducing coupling is in principle quite simple: find a method in a class that interacts with another class through a non-optimal message sequence function; replace the method's implementation by a new implementation that has the same behavior but lower coupling.

More precisely we view the transformation as a 3-step process:

1. Find a method m of a class C that interacts with a class D using a non-optimal message sequence function q. For this it suffices to find two states $s1$ and $s2$ – which we shall call *witness states* – of an object d of D that are equivalent under the equivalence relation induced by m but not equivalent under the equivalence relation induced by q. In other words the two states yield different message sequences for d but they cannot affect the states of other objects differently. We may view such a pair of witness states as an indication that the coupling can be improved.

2. The second step consists in transforming the current implementation into one with lower coupling. The actual transformation will be done via simple refactorings. Which refactorings apply will depend on the case at hand but it will require manual inspection of the code.
3. The third step is optional: it consists in proving that the representational coupling of the new implementation is optimal. It assumes that the solution obtained through the transformation process has indeed optimal representational coupling equal to the intrinsic representational coupling.

We shall demonstrate this 3-step process by applying it to three examples from the literature. We use the following refactorings (in step 2):

- *ExtractMethod*: applied when a fragment of code can be grouped together; consists in turning the fragment into a method whose name explains the purpose of the method.
- *DecomposeConditional*: applied when there is a complicated conditional (if-then-else) statement; consists in extracting methods from the condition, then part, and else parts. Note: this refactoring may be viewed as a special case of the ExtractMethod refactoring where the code fragment is part of a conditional instruction.
- *MoveMethod*: applied when a method is using more features of another class than the class on which it is defined; consists in creating a new method with a similar body in the class it uses most and either turning the old method into a simple delegation, or removing it altogether.

Before we examine the examples, let us consider the relationship of the transformation process with existing work on refactoring. Most tools available for refactoring automate the process of safely applying a refactoring step; an example of such a tool is the Smalltalk Refactoring Browser ([RBJ97]). The problem of detecting candidates for refactoring is more difficult. The usual approach is based on manual inspection of the code to detect "bad smells" ([Fow99]). Only recently has a tool been developed that directly assists the developer in finding refactoring candidates; it is based on detecting invariants ([KEGN01]). Our method provides another way of finding candidates of refactoring that is based on a mathematically precise condition (see Lemma 1). In this sense we consider our approach to be complementary to existing approaches for finding refactoring candidates. Note however that we do not know at this point whether our condition for refactoring can be tested in an automated fashion.

5.1 Example 1: The Elevator Control System

We claim that in the first solution (Implementation 1) the handleRequest() method has non-optimal representational coupling with the Elevator class. To prove this we look for a pair of witness states for an elevator object: if a request is for a floor r other than the first or the last floor, then elevators at position $r-1$ going up and at position $r+1$ going down will be at the same distance

from the request. The position() and direction() queries will return different results but the states of other objects will not be affected differently. Thus the representational coupling is indeed non-optimal.

To obtain a solution with lower coupling, we first apply the ExtractMethod refactoring to the line of pseudo-code that computes the distance. The resulting computeDistance() method only uses features of the Elevator class. We can therefore apply a second refactoring step, the MoveMethod refactoring, to move this method to the Elevator class and delete the original method. By applying these two simple refactoring steps in sequence we arrive at the improved solution (given in Sect. 3.2).

We claim that in this second implementation the handleRequest() method has optimal representational coupling with respect to the Elevator class. We first show that the equivalence relation induced by handleRequest() has as equivalence classes the sets of states that yield the same distance to the request. This can be proven as follows: if two states yield the same distance, then they are treated the same by the state mapping. Conversely, two states with different distances $d_1 < d_2$ from the request can easily be distinguished: simply set the other states to a value that yields minimal distance d_1. Setting the state of the first elevator such that the distance is d_1 will result in it being closest (assume that in case of a tie the first elevator is chosen) and setting it to a state with distance d_2 will result in another elevator being closest.

Finally we observe that the message sequence function in the second implementation (yielding an actual message sequence consisting of the single computeDistance() query and its return value) induces exactly the same equivalence relation as the handleRequest() method. Thus we can apply Lemma 1, proving that the second implementation has indeed optimal representational coupling with respect to the Elevator class.

5.2 Example 2: The Heating System

This example is taken from [Rich99,Riel96] (in slightly adapted form). In a building a number of rooms are controlled by a central heating system. Each room has a current temperature, a desired temperature and an indication of whether it is occupied. The heating system will poll the rooms at regular time intervals and open the valve for the room thus providing it with heat if it is occupied and the current temperature is below the desired level.

In the first implementation there are three classes HeatControl, Room and Valve. The HeatControl class has a single method checkRooms() that performs the previously described polling process. The class Room has three methods *temperature()*, *desiredTemperature()* and *occupied()* returning the current temperature, the desired temperature and an occupancy flag. Finally the Valve class has a single method *command(c)* with an argument c that is either CLOSE or OPEN.

The checkRooms method works as follows: for each room it determines whether the room is occupied and whether the temperature is below the desired temperature by invoking the corresponding methods of the Room class. If

this condition is satisfied then HeatControl sends a open message to the valve for this room, otherwise it sends a close message to the valve.

Here is the pseudo-code for the checkRooms() method in the first implementation:

Implementation 2 *Method HeatControl.checkRooms()*

```
1. for each room r do {
2.     Valve v = r.getValve();
3.     if (r.temperature()<r.desiredTemperature() and r.occupied())
4.         v.command(OPEN);//send heat
5.     else
6.         v.command(CLOSE);//cut off heat
7. }
```

Let C denote the condition of the if-statement. Consider two states of a Room object that have different values for the individual queries in C but yield the same value for C. Two such states form a pair of witness states, showing that the checkRooms() method has non-optimal representational coupling with respect to the Room class.

How do we transform the first solution into one with lower representational coupling? We apply the DecomposeConditional refactoring to extract the condition in the if-then-else statement into a method *needsHeat*(). This method uses only features of the Room class and is therefore moved into that class using the MoveMethod Refactoring. This leads us to the second implementation in which the function checkRooms() asks each room whether it needs heat:

Implementation 3 *Method HeatControl.checkRooms()*

```
1. for each room r do {
2.     Valve v = r.getValve();
3.     if (r.needsHeat())
4.         v.command(OPEN);
5.     else
6.         v.command(CLOSE);
7. }
```

We note that the second implementation is more flexible than the first one: we can for instance change the rules under which a room is heated (e.g., heat only during certain time periods) by modifying only the Room class; in the first implementation both the Room class and the HeatControl class need to be modified.

We can apply Lemma 1 to show that the second implementation has minimal representational coupling w.r.t the Room class: indeed the equivalence relation induced by *checkRooms* on the state space of the Room class is the same as that induced by the message sequence function.

5.3 Example 3: A Stock Trading System

Both of the previous examples are control-oriented systems. This last example taken from [Rich99] is of a different kind. It represents a stock trading system. The system contains four classes of interest to us: ServiceRepresentative, TradingSystem, Order and OrderRegistry. For this discussion we assume that the ServiceRepresentative can cancel an order by invoking the cancelOrder() method of TradingSystem with the order number as argument. The Order class contains a method getStatus() that returns the current status of the order: open or not open (the latter choice applying when the order has been executed, it has expired or has been previously canceled). It also contains a cancel() method. The trading system can only cancel the order if it is open. In the first implementation the TradingSystem.cancelOrder()method first checks the status of the order; if it is open, it issues a cancel() message to the proper order.

Here is this implementation for the cancelOrder() method (we assume that OPEN is a constant of the Order class and TradingSystem.getOrderRegistry() returns a unique object of class OrderRegistry):

Implementation 4 *Method TradingSystem.cancelOrder(orderNumber)*
1. Order o = getOrderRegistry().getOrder(orderNumber);
2. if (o.getStatus()==Order.OPEN)
3. o.cancel();

Consider two states of an order object: open and not open. These two states yield different message sequences for the order object but yet they cannot affect the states of objects other than itself! It follows that the cancelOrder() method has non-optimal representational coupling w.r.t. the Order class.

To arrive at a solution with lower coupling, we first apply DecomposeConditional to extract the condition of the if-then-else statement into a separate method *canBeCanceled*(). That method uses only features of the Order class and is therefore moved to that class. Here is the resulting code:

Implementation 5 *Method TradingSystem.cancelOrder(orderNumber)*
1. Order o = getOrderRegistry.getOrder(orderNumber);
2. if (o.canBeCanceled())
3. o.cancel();

This second implementation is more flexible than the first one: indeed we can change the rule under which an order can be canceled. In Implementation 2 this only requires the implementation of canBeCanceled() to be modified in the Order class while in Implementation 1 the TradingSystem class is also affected since the method *TradingSystem.cancelOrder*() has to modified.

One thing has not changed though: the open and non-open states are still witnesses to the non-optimality of the implementation. Thus we need to look further for an optimal solution.

To achieve minimal representational coupling, we observe that line 2 and line 3 both use only features of the order object and can thus be moved using the ExtractMethod and MoveMethod refactorings to the Order class.

In other words lines 2 and 3 get replaced by a single line

```
o.cancel();
```

The cancel() method of the Order class is changed accordingly. The resulting implementation has obviously minimal representational coupling since the actual message sequence is independent of the state of the Order object.

6 Conclusions

In this paper we have proposed a mathematically precise definition of representational coupling based on the information exchanged between a method and a class via messages. We have also defined the notion of minimal representational coupling which expresses the minimal amount of representational coupling inherent in an implementation. Finally we explain, using several examples, how these notions can be used to identify candidate methods for refactoring and how they can guide us in the refactoring process.

There are quite a few open questions that need to be examined:

- Can we apply the transformation process described in the previous section to real-life examples consisting of several hundred or even thousands of classes? For this it would be helpful to automate the process (as it is done for software metrics). Because of the non-quantitative nature of our measure it is not clear whether this is possible.
- So far we have only considered the representational coupling with a single class. Can our approach be generalized to analyzing the representational coupling with several classes at once? Since a method of a class interacts typically with several other classes to perform its function, this is of course highly relevant if we want to apply our process to large software systems.
- What is the exact relationship with refactoring? Can a method with non-optimal coupling always be refactored to one with minimal coupling?

References

[Ari02] Erik Arisholm, Dynamic Coupling Measures for Object-Oriented Software, Proc. of the Eighth International Symposium on Software Metrics (Metrics'02), Ottawa, Canada, pp. 33–42.

[BBM96] V.R. Basili, L.C. Briand, W.L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators, Transactions on Software Engineering 22(10), pp. 751–761, 1996.

[Bol95] T. Bollinger, "What Can Happen When Metrics Make the Call," Transactions IEEE Software, vol. 12, no. 1, Jan 1995.

[Bri97] L.C. Briand, P. Devanbu, and W. Melo, An investigation into coupling measures for C++, Proc. 19th Int'l Conf. Software Eng., ICSE'97, Boston, pp. 412–421, May 1997. .

[Bri99a] L.C. Briand, J.W. Daly, J.K. Wüst, A Unified Framework for Coupling Measurement in Object-Oriented Systems, IEEE Transactions on Software Engineering, vol. 25, No. 1, Jan/Feb 1999.

[Bri99b] L.C. Briand, J.K. Wüst, H. Lounis, Investigating quality factors in object-oriented designs: an industrial case study, Proceedings of the 21st international conference on Software engineering, pp. 345–354, 1999, Los Angeles, California, United States.

[Cha93] D. de Champeaux, D. Lea, P. Faure, Object-Oriented Systems Development, Reading, Mass.; Addison Wesley, 1993.

[ChiKem91] S.R. Chidamber and C.F. Kemerer, Towards a metrics suite for Object Oriented Design, Proc. Conf. Object-Oriented Programming: Systems, Languages and Applications, OOPSLA'91, Oct. 1991. Also published in SIGPLAN Notices, vol. 26, No. 11, pp. 197–211, 1991.

[ChiKem94] S.R. Chidamber and C.F. Kemerer, A metrics suite for Object Oriented Design, IEEE Trans. on Software Engineering, vol. 20, No. 6, pp. 476–493, 1994.

[Eder94] J. Eder, G. Kappel, and M. Schrefl, Coupling and Cohesion in Object-Oriented Systems, Technical Report, Univ. of Klagenfurt, 1994.

[Fow99] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[HiMo95] M. Hitz and B. Montazeri, Measuring Coupling and Cohesion in Object-Oriented Systems, Proc. Int'l Symp. Applied Corporate Computing, Monterrey, Mexico, Oct. 1995. A version of this paper (focusing on coupling only) has been published in *Object Currents*, vol. 1, No. 4, SIGS publications, 1996.

[Jon99] C. Jones, "Software Metrics: Good, Bad, and Missing," Computer, vol. 27, no. 9, Sept 1994.

[KEGN01] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin, Automated support for program refactoring using invariants, ICSM 2001, Proceedings of the International Conference on Software Maintenance, Florence, Italy, November 6–10, 2001, pp. 736-743.

[Lee95] Y.S. Lee, B.-S. Liang, S.-F- Wu, and F.-J. Wang, Measuring the Coupling and Cohesion of an Object-Oriented Program based on information flow, Proc. Int'l Conf. Software Quality, Maribor, Slovenia, 1995.

[LH93] W. Li and S. Henry, Object-Oriented Metrics that Predict Maintanability, J. Systems and Software, vol. 23, no. 2, pp. 111–122, 1993.

[Rich99] C. Richter, Designing Flexible Object-Oriented Systems with UML, Macmillan Technical Publishing, 1999.

[Riel96] A. Riel, Object-Oriented Design Heuristics, Reading, Mass.: Addison-Wesley, 1996.

[RBJ97] D. Roberts, J. Brant and R. Johnson, A Refactoring tool for Smalltalk, Theory and Practice of Object Systems, 3(4), pp. 253–263, 1997.

[Yac99] Sherif M. Yacoub, Hany H. Ammar, and Tom Robinson, Dynamic Metrics for Object-Oriented Designs, Proc. of the sixth International Symposium on Software Metrics (Metrics'99), Boca Raton, Florida, USA, pp. 60–61.