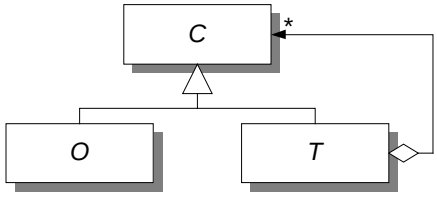
Object Relational Modeling

COT/4-04-V1.0



Centre for Object Technology

Centre for
Object Technology

Revision history: V1.0 First public version
Author(s): Allan R. Lassen, Rambøll informatics
Johnny Olsson, WM data
Kasper Østerbye, Aalborg University
Status: Public

Publication: Public

Summary:

In the database area, there has been some pressure to support richer semantic models than the now classical relational model. The pressure has been to support the object-oriented model. However, the larger market has not been willing to adopt a pure object-oriented model, but some of the major producers of database management systems have introduced systems which they call *object-relational*, trying to combine relational and object oriented technologies. This report embraces that approach, and takes the perspective that relations are a natural thing in modeling reality. Our notion of *object-relation*al modeling is borrowed from Rumbaugh. The report describes the fundamentals of the model, it reports on experiments in implementing it in both an object-oriented language (Smalltalk), and a modern *object-relation*al database (Oracle 8). It gives a brief outline on some preliminary experiences from a concrete modeling case, the rule-checker of the STADS system, and it points to some new directions which must be investigated.

The Centre for Object Technology (COT) is a three year project concerned with research, application and implementation of object technology in Danish companies. The project is financially supported by The Centre of IT-Research (CIT) and the Danish Ministry of Industry.

Participants are:
Maersk Line, Maersk Training Centre, Bang & Olufsen, WM-data, Rambøll, Danfoss, Systematic Software Engineering, Odense Steel Shipyard, A.P. Møller, University of Aarhus, Odense University, University of Copenhagen, Danish Technological Institute and Danish Maritime Institute

CONTENTS

1 INTRODUCTION

In recent years object oriented techniques have moved from being primarily a programming paradigm to also include earlier phases of software development. The most prominent notation used in object-oriented analysis and design today is UML. UML introduces new abstraction mechanisms, which were not previously part of the object oriented programming paradigm. Most noteworthy is the concept of *association* and *aggregation*. An important claim to fame for object oriented techniques is that the same abstraction mechanisms are used throughout analysis, design and implementation. Hence, it seems strange that associations should be supported only during analysis and design. This leads to a semantic gap between design and implementation. In 1987, Rumbaugh published a paper [2], in which he argues that associations (in that paper called relations, a naming we adopt in this report as well) are more important to modeling than specialization. He also demonstrates how relations can be supported directly in programming languages and give proof of concept through the language DSM [3].

This report will try to verify Rumbaugh's claims. This will be done at two levels. First, we will carry out the experiment to incorporate object-relational modeling techniques in the object oriented programming language Smalltalk. Smalltalk is chosen for two reasons, it serves as a test to see how well relations can blend into a language, which is considered to have a "pure" object oriented model, and it is relatively simple to accommodate new constructs through meta-class programming. Parallel to the Smalltalk implementation, we will examine how well a modern object-relational system supports Rumbaugh's ideas. At the second level, we have tried the object-relational modeling technique on a larger system, the STADS system developed by WM-data.

1.1 OUTLINE OF REPORT

The report is in three major parts. Sections 2, 3, and 4 deals directly with the model for object-relational programming described by Rumbaugh. First, his model is described in section 2, in section 3 we describe our experiences in creating support for the object-relational paradigm in Smalltalk and Oracle 8, representing two very different approaches. Finally, in section 4 then we present and comment on our experiences on using object-relational model on a large system. Section 5 represents the second part of the report, and deals with further issues which are needed to consider as extension and clarifications of the object-relational model. These issues arose from our experiences in doing object-relational modeling in our case study. The last part of the report is the appendices 7 and 8, which describe how to implement the object-relational model in Smalltalk and Oracle 8. The choice of these two systems is primarily based on our own interest, and they each have some appealing features. Smalltalk is quite flexible because of meta-classes, and the system can be changed to assimilate the object-relational model nicely. Oracle 8 claims to be "object-relational" from birth, and it is therefore interesting to examine how well one can get to Rumbaugh's model.

2 THE OBJECT-RELATIONAL MODEL OF RUMBAUGH

As described earlier, the main goal of this paper is to verify the results presented by Rumbaugh. If this is possible, we believe that relations should play a much more prominent role in object-oriented programming than it does today.

2.1 THE MODEL

The programming language DSM [2] includes relations as a semantic construct. In this section, we will present the most important aspects of the model, and some of its shortcomings as pointed out by Rumbaugh.

The fundamental idea is to introduce relations as a language construct at same level as classes. Just as classes describe a set of objects, relations describe associations *between* objects. Between is the keyword here, a relation does not belong to either of the classes it connects, but is something that ties different classes together.

In DSM a relation can be declared as:

```
RELATION Works_for
        (employee: Person, employer: Company)
```

This states that there is a relationship between objects of type Person and Company. That the relationship is named Works_for, and that the objects of type Person plays the role of employees in the relationship, while Companies play the role of employer.

To add concrete associations to the Works_for relation, the following syntax can be used:

```
Works_for.add(Allan, Rambøll)
Works_for.add(Kirsten, WM_data)
Works_for.add(Johnny, WM_data)
Works_for.add(Kasper, Aalborg University)
```

Two types of query are supported in DSM. One can ask if two objects are related through the test operation on relations:

```
Works_for.test(Allan, Aalborg University) returns false
Works_for.test(Johnny, WM_data) returns true.
```

To get the set of objects related to a given object the DSM model uses a bizarre syntax:

```
Works_for.index_2(WM_data) returns {Kirsten, Johnny}
Works_for.index_2(StoneWare) returns {}
```

A more readable syntax would be:

```
Works_for.getEmployees(WM_data) returns {Kirsten, Johnny}
Works_for.getEmployees(StoneWare) returns {}
```

The model also supports cardinality constraints and a simplified kind of three-way relation called a qualified relation. The model is further discussed in [2,3].

2.2 SUPPORT OR SIMULATION

An important question to consider is whether it is necessary to introduce a new language construct. This becomes an especially important question in the light that most object oriented analysis and design notations have had support for associations, and programming languages have not. Rumbaugh mentions six reasons why it is necessary to let relations be a language construct in its own right: (Near verbatim citation from sec. 3.3 in [2])

Information hiding. The language construct should not reveal its implementation. There may be more than one possible implementation of a logical relation. A programmer should be able to choose the implementation using an option flag on the declaration, without changing the code that uses the declaration or even most of the declaration itself.

Initialization. A more technical issue is that the compiler can instantiate and initialize relations implicitly at the beginning of program execution, just as object classes are instantiated and initialized.

Special syntax. In addition, the language construct should have a syntactic appearance that makes it feasible and natural to use. The language can provide special syntax to simplify operations on relations, just as special syntax is provided for method application.

Access methods. The compiler can provide syntactic sugar to access and update relations seen from the view of the participating objects. The compiler can automatically generate methods on the participating object classes to access and update the relations.

Integrity. There must be built-in means for managing relations and objects when objects are destroyed. Object classes will have a list of relations they participate in, represented in a uniform way. This information can be used in writing generic methods to destroy objects and clean up relations they participate in, to copy objects and objects they are related to, and to pretty print objects along related objects.

Conceptual support. Finally, there is an important aspect of non-technical type. Most importantly, treating relations as important built-in semantic constructs changes the way programmers visualizes and formulate problems. Thinking in terms of objects and generalizations hierarchies is in generally unfamiliar at first, but eventually changes the way programmers thinks about a problem. We have found from experience that making relations a first-class semantic construct affects a programmer's way of thinking about a problem from the design stage all the way through the coding. This new way of thinking is particularly useful for formulating and partitioning designs.

We are quite compelled by these arguments, but we think that the idea of relations in object-oriented programming needs further empirical grounding.

2.3 RELATED WORK

Apparently, there has been much work following up on Rumbaugh object-relational

modeling and giving semantic support in programming languages. Rumbaugh himself has since only considered the modeling part through OMT and later UML. Giving semantic support in programming languages have been given very little attention in the research community as far as we can tell. However, we can point to two papers, which addresses the issues. In [6] March and Rho describes an object-relational system, which adds E-R semantics to object oriented modeling. Their system is very similar to the Smalltalk system we have developed (see appendix 7). It is a direct implementation of Rumbaugh's model, with some extensions for querying and navigation. As is done in our system, March and Rho uses meta-classes to implement access methods etc. One difference is that we implement relations as classes whereas they realize them as objects. At the modeling level, we support attributes on relations, which they do not. On the other hand, they have gone further by supporting queries for relations as well as for classes.

As mentioned, the notion of association has had a strong influence on the analysis and design notations, but has not had any impact on programming languages. This leaves us with a serious question of why that is the case. Three possible answers present themselves. (1) Supporting associations is a bad idea. (2) There is a gap in the design of object oriented programming languages. (3) The notion of associations is not well matched to object oriented modeling. Our standpoint has been that (2) is the case. In [1] Velho and Carapuça argues that (3) is the case. They present three arguments to support the claim. First, relations break down class encapsulation because relations are external to the objects, but state something about the objects anyhow. Second, they state that relations break down abstraction because relations become a kind of entity without real world counterpart. Their argument is that relations introduce a semantic gap between the real world and the created models. Lastly, they argue that relations break extensibility because an inheritance mechanism will be needed for relations, and relation inheritance will most likely differ from that of classes.

We find that all three claims can be refuted. Their last argument is likely to be correct, but under other circumstances, such an argument will be put on the agenda for further investigation, as indeed it is done in Rumbaugh's original paper [2]. The first two arguments are really a fundamental issue on the nature of object orientedness. They take for granted that the fundamental abstraction is objects *and only objects.* In the Scandinavian school of object oriented programming, there is a deeper foundation, which is that the fundamental issue is modeling the real world using natural abstractions. If relations turn out to be natural to use, modeling should support this, and so should programming to avoid a semantic gap between model and program. We are convinced that relationships are a natural abstraction. However, the ideas presented by Velho and Carapuça are an interesting alternative to relations, and it would be an interesting exercise compare the two approaches on the same underlying model, (e.g. the model introduced in the next section).

3 IMPLEMENTATION STRATEGIES

This section will examine two strategies for implementing an object-oriented model with relations. First, we will address how to extend existing object-oriented languages with relation support, as well as examine how relations can be mapped into standard object oriented languages. Secondly, we will investigate implementation in the new object-relational paradigm, exemplified by Oracle 8. These two discussions will loosely follow the six issues raised by Rumbaugh in section 2.2.

3.1 OBJECT ORIENTED PROGRAMMING

For the programmer there are two ways to come from an object-relational model to a purely object oriented model. The easiest would be if the compiler were extended to include relations as a semantic construct, letting the compiler take care of the translation. The harder way is to transform the relations into classes, objects and methods by hand. This section examines both strategies.

Binary relations between objects can be seen as a set of pairs of object identifiers. The relation can then be seen as a collection object, which has methods for adding and removing pairs. The collection must also provide methods for querying, allowing us to find all the objects related to a given object. It is therefore straightforward to implement relations in an object-oriented language. However, this implementation does not meet all the requirements put forward in section 2.2.

The simple solution offers some degree of information hiding, the relation as a collection object does not reveal its implementation, whether it stores the pairs in a set, or in hash tables, or by some other means. In practice, one would properly provide several implementations to support different categories of cardinality, a 1-1 relation can be implemented more efficiently than a many to many relation. However, the relation does reveal itself as an object. Initialization of the relation can be done in the constructor of the relation collection, initializing the relation only when needed. One can argue that this is more appropriate than the compiler forcing it to happen at the beginning of the program execution as proposed by Rumbaugh. This approach allows non-global relations, which is not possible in DSM. Regarding the issue of special syntax, the “relation as collection object” obviously cannot provide specialized syntax, as this requires changes to the compiler[1]. The issue of special access methods is a particular case of specialized syntax. The issue of integrity is hard to address properly without compiler support. The relations the object participates in must be updated when the object is destroyed. There is no inherent problem in doing this, but without compiler support there is no guarantee that it will be done.

The last, and perhaps most important, issue is conceptual support. Here it is very difficult to say what will happen if one stick to the simple collection object solution to relations. One can argue that there should always be specialized syntax to support important semantic constructs. The language SETL proposed specialized syntax for Sets, APL had specialized syntax for matrix calculations. However, it has been the experi-

[1] Alternatively, it can be done using Meta programming as discussed in section 7.

ence of Smalltalk programmers that the standard collection classes, has a deep impact on their programming style, even though no special syntax is provided. Similarly, operator overloading in C++ allows very elegant solution to matrix calculations without the cost of specialized syntax. Whether or not the simple “relation as collection class” solution is sufficient depends as much on the syntactic elegance of the programming language in question – e.g. how well a the relation-collections can masquerade as proper relations - as it does on general properties of object oriented programming.

However, we believe that the issues of special syntax, especially access methods, and integrity does warrant compiler support. The question is then how much work need to be done by the compiler to support relations. This will depend on the programming language in question. In our experience with Smalltalk, there has been no need for special syntax beyond access methods, and we believe that to be the case in general. We have not investigated to what extend relations can be implemented conveniently using pre-compilers or if they need to be tied closely into the compiler proper.

Integrity is a much harder problem, since it involves the details of instantiation and destruction of objects. In a few cases, the relationship is constrained in such a way that any objects of a specific type X *must* be related to another object of type Y. This means that whenever a new X-object is created, it must be related to an Y-object. To ensure this the constructor of X must be changed to address this. However, it is not apparent how this can be done in general, and it seems unlikely to be a subject for automation. When an object is destructed, it must be removed from its relations. Again, this is a language specific issue. In languages without garbage collection, the object must be removed from the relations in order to avoid dangling references. If the language does support garbage collection, the programmer must be aware that the relations will prevent the object from being collected.

An alternative to both the “relation as collection object” and semantic support through compiler support, is to follow current programming practice of programming of transforming associations and aggregation structures based on UML specifications. This practice is characterized by hand-coding the associations according to the specific properties of the specified associations. To the experienced programmer, this is normally done using references and collections, and the process of establishing navigation paths between objects is a natural and important program design issue. E.g. a programmer might realize that a specific one-to-many association between X and Y is really only mono-directional, and therefore implements the association as a set in X containing all the Ys it is associated with. Such transformations should be possible to formalize into a pattern language and design patterns. For example, a 1-to-1 relation can be implemented as a pair of mutual references in the participating objects under most circumstances. The advantage of such transformation is that the programmer is better at choosing the optimal solution for each specific situation, and that no special support is needed. As earlier said, this is current practice, so no one is confused by a paradigm shift.

3.2 OBJECT-RELATIONAL MODELING, ORACLE 8

In an object-relational database environment like Oracle 8, implementing an object-oriented model with relations involves defining the model using the data definition lan-

guage from SQL and implementing the functionality of objects, i.e. methods, through the PL/SQL language, which is Oracles procedural extension to SQL. The model can then be accessed from the environment itself e.g. with Oracle SQL*Plus using SQL and PL/SQL.

Using embedded SQL in languages like C/C++ or Cobol you can execute any SQL statement including data definition statements from an application program. However, here is no difference in implementing the model in SQL*Plus or using a pre-compiler like Oracle Pro*C/C++. In both cases, SQL and PL/SQL will be used to define the data model.

Alternatively, one could consider implementing Rumbaugh's model in C++ using Oracle 8 as the persistent store. This has not been further investigated because we believe it will lead to something close to the Smalltalk implementation. Therefore, in our Oracle 8 implementation we will stick with plain SQL and PL/SQL.

The Oracle 8 environment naturally supports relations but the implementation does not meet the all the requirements from section 2.2. In a way, a lot of the discussion from the previous section also applies to Oracle 8. The SQL language i.e. the data definition statements could be extended to support the missing features and making the compiler responsible of generating the needed code. Another possibility is to develop some kind of pre-compiler that will transform the extended SQL statements to the statements that is currently supported in SQL and PL/SQL. Finally, the programmer can hand-code it from general patterns as discussed in the previous section.

In our implementation in chapter 8, the issue of information hiding is almost given in advance, because we have to use database tables to implement many-to-many relationships. We would then hide these tables by defining proper access rights so they would only be accessible to the public through the implemented package. An important point is that we can choose to implement one-to-one and one-to-many relationships without using a database table and still keep the package interface intact. It is therefore possible to achieve information hiding.

We found no need for initialization of the relation. We do have the opportunity though in the package. A package contains a section that functions like a constructor in object oriented languages, and this section is executed when the package is first used.

The issue about special syntax is fulfilled in the sense that the relation is implemented in a database table and a package. Therefore, we can use relations the same way as we use classes and we believe that is as close as we can get to natural use in the programming paradigm. We have to provide the access methods ourselves but we could develop a pre-compiler that will automate the process. We do not believe that we need changes to the syntax to address the issues of special syntax and access methods. In our experiments, we found that we could not use the access methods in all contexts, so we had to code it in a different way. Thus, we would desire better support of using the methods in SQL-statements.

At present, we cannot solve the issue of integrity with built-in features for references to objects. If a referenced object is destroyed, the reference will be dangling. Oracle,

however, does provide facilities to ensure referential integrity on foreign keys in the relational database, namely by declaring constraints. As a relational database, Oracle 8 still supports foreign key constraints. Nevertheless, our problem is that we use the new object references and they lack constraint facilities. We expect they will be implemented in a later release of the Oracle database. Meanwhile we may use database triggers for delete operations on the tables. The triggers could be generated automatically.

It is interesting to see how programming will change when we focus equally on classes and relations. Today we usually try to eliminate the relation itself and implement its function in the participating objects. One can argue that programming always will be based on the object classes and that we will use their access methods to get to their related objects. Perhaps that will often be the case because the relation doesn't have a strong and natural position in the problem domain. This may change however if we name the relations by the noun form of verbs instead of plain verbs. In Rumbaugh's example with the Works_for relation it may seem awkward to express Works_for.Add(...). Instead we could name the relation Employment and the statement will then be Employment.Add(...) which is easier to cope with. We believe that kind of relation names give a more natural way of thinking about a problem and if we have problems finding a proper name then it is probably because the relation is weak in the problem domain.

4 THE STADS RULE CHECKER – A CASE STUDY

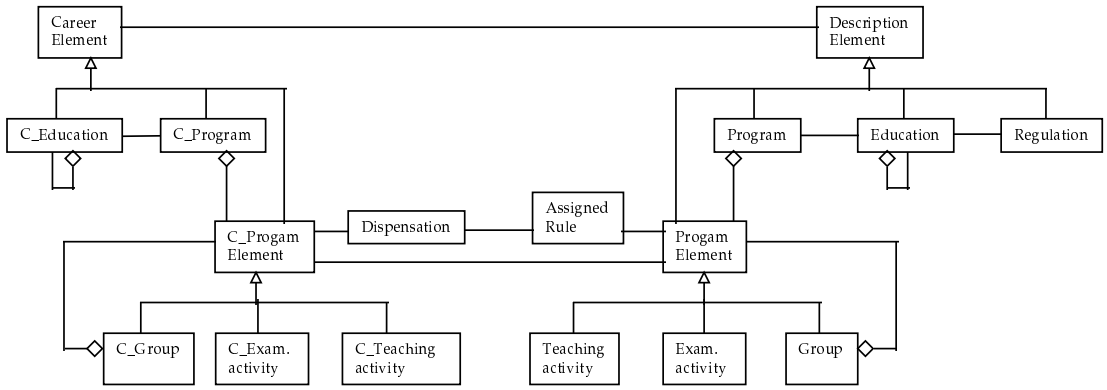
To test relations as a semantic construct, we have used them in a paper and pencil example, combined with a simple usage of the Smalltalk implementation described in section 7. We are aware that this does not provide us with the same level of understanding as a major programming project would have provided. However, several interesting observations have been made.

The case deals with one aspect of a large generic administration system called STADS[2]. The selected part deals with describing educations and the courses and exams that constitute these educations. It deals with describing the concrete career of individual students, and most importantly, it deals with describing the rules that make up the passing criteria and structural constraints of the educations themselves. The analysis and design of the STADS system is further described [7].

To provide a framework for describing educations, a number of concepts have been defined, ranging from government regulations and degrees, to courses and individual exams. Each of these concepts can be described using rules. For an individual exam, the rule might say something about the passing grade, for a course it might say something about the minimum grade average needed for its constituent grades. The rules associated with a degree might state that a certain number of courses must be passed, or that in general all exams must be passed with a certain minimum grade. The career of a specific student contains the courses actually followed by the student. Most educations are structured to have required as well as voluntary courses, and the career of each student therefore varies in structure as well as in grade results, even if they might end up with the same degree.

Relations have played an important role in describing the STADS rule checker. They have been used to describe the relation between the different descriptive elements (degrees etc.), the parallel relationship between the corresponding career elements, and to associate career elements with their descriptions. Relations are also used to associate descriptive elements to their rules, and to associate individual dispensations from the rules with career elements for the student who has the dispensation. The following figure gives a fuller picture of the model.

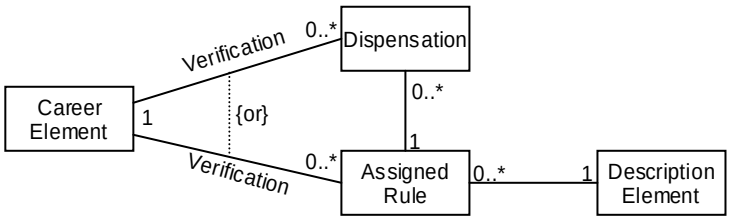
[2] STADS is a short for 'STudieADminstrationsSystem' (Education Administration System in Danish). The system concerns administration of students, student programs, teaching, exam planning, and public scholarships. STADS has been developed since 1993. The first universities started to use the system in 1996. As the system gets more developed, more universities are taking in into use. The system is implemented on a Oracle 7 platform and works for UNIX as well as VMS



In the diagram, there is a single association between Career element and Description element. This association should be interpreted as a pair-wise association between specializations within the two hierarchies. As an example it is interpreted as associating Career programs (the program followed by a specific student) to programs as offered by the educational institution. The association is repeated between the Career program element and the program element.

It is worth to notice that the career elements are *not* instances of the descriptive elements. The reason is that the descriptive elements are tangible objects in the domain of the system. The users of the system are used to having such things as course descriptions in their problem domain. This is a prime example of the object-oriented analysis pattern “item-description” described by Coad [4].

The hard problem however, is the design of the rule checker. The rule checker can check if the career of a specific student follows the rules of the associated descriptions, taking into account rules from general descriptions, and taking into account any relevant individual dispensations. The relationship between a career element and relevant rules and dispensations are called the *verification* relation.



This relation is to costly to maintain, and must be computed when needed, this is called a derived relation, and it is an issue which will be further elaborated in section 5.3. So far we have only seen a need to find the rules and dispensations for a given career element, not finding all the career elements cor a given rule. The computation of the verification relations is as follows. For a specific career element, the relevant rules are obtained by first finding the corresponding descriptive element, and then obtaining the rules from there. Then we find the father of the descriptive element (e.g. an exam might be part of a course, which might be part of a degree, which might be under government regulations), and the rules from the father is added to the effective set of rules. If an assigned rule is encountered twice, only the most specific assigned rule counts.

Finally, all rules with dispensations are removed, and the dispensations are added instead.

4.1 EXPERIENCES

Our experiences can be summarized as: Relations are well suited to describe simple associations between objects, Relations are well suited to understand complex associations between objects, but it can be hard to formalize complex relations between objects.

We used the Smalltalk implementation of relations to implement all the associations and aggregations used in the UML diagrams from the analysis model. It was a very rewarding experience to see the direct translation and direct connection between code and diagram. A practical matter arose though, which was one of naming. Each association on the UML diagram must be given a name to be implemented. To come up with meaningful names for each association was very hard, and in the end we just settled for a simple systematic naming involving the first part of each participating class. We believe, however, that the problem has a more profound explanation. Throughout the work with the STADS system, we were lacking a proper object that captured the structure of a program or career. Many of the relations in both career and description are structural relations, and a compelling idea is to interpret these unnamable associations as making up a single relation (one for career and one for description), which exactly defines the structure.

However, this leaves us with a typing problem. E.g. a program element can be part of both a program and a group (which is a modeling concept used to provide some reuse of rules and descriptions).

The Smalltalk implementation compiles access methods into all participating classes. This eliminates the need to refer to the relation by relation name, but allow us to move from one object to the other by the role names specified as part of the relation definition.

When the details of the rule checker was defined, thinking in terms of relations proved valuable. Rather than focussing on the implementation of the checking algorithm, it lead us to consider its specification in terms of defining the verification relation between the career element on the one side, and assigned rules and dispensations on the other side. The discovery of this relation proved to be pivotal to our understanding of the inner working the rule checker. However, defining the exact semantics of the verification relation proved hard. The issue of derived relations is further discussed in section 5.3 below.

5 ISSUES

In this section we will present some issues that has presented themselves in our experiments and some issues which we find interesting and think deserves further investigation. The section is unstructured, consisting of a set of independent issues.

5.1 SUPPORTING UML, UML SUPPORT

Introducing relations into object oriented programming can provide a seamless transition from design notation to implementation. This can be done because relations can support the different kinds of associations that can be expressed in UML, the most widespread design notation in use today. Associations in UML cover both aggregation structures as well as more loose associations. We find that it is very interesting, and not outside the scope of a modern compiler to provide different implementations depending on the kind of association and its annotations. On the other hand, we our experiments have revealed that relations are not quite the same as UML associations. Relations, as discussed so far, are named, and have associated role names for the participating objects. Role names in particular seem to be the corner stone in the seamless integration of relations into the object oriented paradigm. The role names become the name of the access methods in the participating classes, allowing method invocation to be the way to query a relation. It is therefore important to use the full UML notation to state associations, specifying both the name of the association as well as its role names.

5.2 OBJECT ORIENTED PROGRAMMING WITHOUT REFERENCES

So far we have been discussing the issue of merging relations into the object-oriented paradigm. A more radical approach is to let relations completely replace references in object oriented programming. We believe this to have several advantages:

- Higher abstraction level. References are one of the few semantic constructs, which have not successfully been subject to abstraction, but is directly inherited from machine code.

- Fewer errors. Using relations rather than references allow us to perform constraint checks, the notion of *sharing* is simply described as an object being related to several object, and variables are only used to model attributes or static components.

- The compiler can produce access methods and choose appropriate implementation strategies depending on the type of relation. This will produce safer code than what is normally produced by human translation, reference errors are well known and can be hard to debug.

- In present object-oriented programming languages, aggregation is only supported through static references. In the object oriented analysis and design notions aggregation has evolved to become a richer notion, which have no direct support in programming languages. Relations might be an answer to this.

- If the compiler can recognize the special case of a “one way, 1-1 association”, then

it can *implement* this as a reference, but then references become an implementation technique used only by the compiler.

As a final comment, it might be that explicit relations can provide a solution to the problem of schema evolution in object oriented languages. References carry little semantic information for the compiler to work with - relations carry a lot.

5.3 DERIVED RELATIONS

In the STADS rule checker the relationship between a career element and a set of assigned rules and dispensations (known as the verification relation) is too expensive to maintain. Rather it must be computed on demand. This leaves us with two issues. How can we define such a derived relationship, and how can we then effectively compute it when it is needed? Our experience was that the very notion of introducing the verification relation was pivotal to our understanding of the algorithm, although we could neither specify nor compute it directly. An obvious path to investigate is to examine query languages for object-relational models, as the verification relation can then be expressed as a derived relation specified by a query.

5.4 RELATIONAL OBJECTS OR EXTENDED TUPLES

In typical relational systems, relationships are values not objects. This means that two objects can only be related to each other in the same relation once. If we incorporate relations as a semantic construct, as proposed in this paper, we need to carefully consider this notion. In object-oriented programming objects are not values, but have unique identity independently of the value of its attributes. It would be natural to adopt this object-oriented view also for relations, making each association unique. This becomes especially interesting in the situation where associations can have attributes. An example is the works-for relation, which naturally can be attributed with salary and job title. If we use the value approach, a person cannot work for the same company in two different ways, e.g. as both driver and night guard. However, at present we have not researched the differences, but it might be that the object view is only necessary when the relations are attributed, and then also only in certain cases. Rumbaugh chooses the value approach, which is interesting, as his example is indeed the works-for relation. This is also the implementation chosen in our Smalltalk implementation.

5.5 INHERITANCE

There are two different aspects to be considered when combining inheritance and relations. The semantics of relations and class inheritance must be resolved, as class inheritance is an integral aspect of object-oriented programming. However, it is interesting to examine the possibility of inheritance for relations as well.

Relations must be defined with such semantics that they work well with class inheritance. In typed object-oriented languages a subtype can normally replace its super type. Defining a relation R between objects of types A and X therefore defines a relation between each of their subtypes. If B is a specialization of A, and Y of X, then R can relate A-objects to Y-objects, B-objects to Y-objects etc.

In the STADS system, we found a need for co-variant specialization of relations, as the relation associating career elements to descriptive elements is specialized to say relate career program elements to descriptive program elements. Translated into general terms, using the A and X classes from above, we see a need to specialize R in such a way that B objects relates to Y object only. The need arises with typed programming languages; when navigating the R relation from a B object, it is most convenient that the returned objects are of type Y, and not X. This prevents intensive type casting. We did not find an immediate need for general techniques for specialization of relations. In fact, the case arose from a example of “parallel inheritance”, where two class hierarchies match each other element by element, and the top classes were abstract. This gives a situation where a general object is never related to a specific one. This implies that the co-variant specialization of relations is only a typing issue, not a query issue. Whenever a B-object is in the relation, it is always related to a Y-object. However, further experience is needed to conclude if other types of relation specialization are needed, and how widespread the need for co-variant specialization is.

6 CONCLUSION

The resent efforts of the database community to provide some support of objects has been scorned by the object oriented community because of inappropriate support for objects, classes, inheritance, polymorfic methods etc. Our investigation of the classic paper by Rumbaugh has verified his claims that relations can complement the object-oriented paradigm well. We have verified that this can be done both at the modeling level, and at the programming level, both through compilation techniques, and by translation into both pure object oriented programming languages and into the object-relational model underlying Oracle 8. Our conclusion is therefore that the object-relational model should not be seen as a steppingstone on the way from relational to object oriented models, but rather as the synthesis of two strong models. However, the conclusion is not that the synthesis has been finished. A number of both practical and theoretical issues remain to be solved - in both modeling and programming.

In the Smalltalk implementation of the STADS model we do not refer to the relations directly, but navigate through roles. We can see two explanations for this: First, it might be that we did not properly identify the meaningful relations in the domain. We have used UML, and as discussed in section 5.1, UML normally do not raise associations to concepts, but let them remain unnamed associations. For instance, many of the associations between descriptions represent a single relation, which we might call *educational structure*. Another explanation is that roles, not the relations themselves, are the important thing to consider in object-relational modeling. At the practical level, roles are perhaps the glue that makes relations become a natural ingredient in the object-oriented paradigm.

One can also ask if relations will always disappear in practice. At a practical level, wrong naming of relations might render them unnatural to use. As an example from section 3.2, the Works_for relation is renamed to Employment, which changes adding of associations from Works_for.Add(...) to Employment.Add(...). However, we have no experience to conclude anything here. A more theoretical issue is whether relations with attributes, e.g. a salary attribute on the employment relation, will also disappear into roles, or if attributed relations will be more natural to use. Other practical issues also present themselves, though they have not surfaces in the examples. For instance, is “total_salary” a method in the company class or on the employment relation?

We have seen a need for derived relations as discussed in section 5.3. The open question is how to specify them. One obvious solution will be to use a query language. In the case of the verification relation from the STADS system, the relation is inherently recursive, which makes it hard to express in most query languages. On the other hand, specifying the relation in the underlying programming language as we did is certainly not the right solution. Rumbaugh also raised the issue of derived relations in his original paper.

Two other interesting issues arose during the STADS work: structural relations and inheritance of relations. Inheritance and relations presented itself in the relation that holds together career elements and description elements. While the relation can be specified at the general, it is tedious to specify that a course description element is re-

lated to a course career element, that an exam description element is related to an exam career element, etc. The problem of structural relations presented itself in the form of the two relations that holds together the career and description elements in a hierarchy.

An important part of the work presented in this report is contained in section 7 and 8 reporting on concrete implementation experiments of Rumbaughs model in Smalltalk and Oracle 8. The experience from these two experiments is summarized in section 3, and will not be restated here.

We have not been able to find much relevant literature on the subject of object-relational modeling or programming. We find this both disturbing and encouraging. It is disturbing because it might indicate that the idea was not useful after all, but encouraging because not much work has been carried out in the field, and hence it should be easy to thread new ground.

Our plans for the spring 1998 are to carry out two parallel projects. At the experimental level, we will continue with the STADS experiment, designing and implementing parts of the model in Oracle 8 and its associated languages, using the recommendations and experiences presented in this report. At the theoretical level, we will study the idea of eliminating references from object-oriented programming languages altogether. This theoretical project will draw upon the practical problems from the STADS experiment, and the STADS experiment is expected to draw upon the preliminary results from theoretical project.

7 SMALLTALK IMPLEMENTATION

We will design a simple relation concept for Smalltalk, based on the ideas of Rumbaugh. The main idea is to let relations be classes, with the *set* of associations living in the meta-class, and the associations themselves being instances of the relation. If we do this, we can quite naturally write expressions in the following way, inspired by some of Rumbaugh's examples:

```
WorksFor testMember: 'Jim Jones' and: 'Acme Products'
```

Here WorksFor is a relation class, which is send the message testMember:and:, with arguments "Jim Jones" and: "Acme Products". This creates a new association object (instance of WorksFor), which is added to the set of associations maintained by the relation WorksFor. Other operations on relations can be implemented similarly as class methods. Further, by letting relations be classes, each class can have its own methods, which allow us do add specialized methods utilizing the role names of the relations to improve readbility:

```
WorksFor employee: 'Jim Jones'
```

The general idea is to define a class Relation, from which concrete relations inherit their behavior. The implementation does rely strongly on meta-classes and the dynamic properties of Smalltalk.

In section 7.1 and 7.2 we address how to create and access relations, and section 7.3 explains the internal representation and the usage of meta-classes in greater detail.

7.1 CREATING RELATIONS

In Rumbaugh examples relations are created in the following way:

```
RELATION Works_for (employee: Person, employer: Company)
```

The following Smalltalk syntax would be similar, while staying within the syntactic limitations of Smalltalk:

```
Relation named: #WorksFor
        relating: #(employee Person)
        to: #(employer Company)
```

A more general message that will create a new relation is of the following form:

```
Relation named: #WorksFor
        relating: #(employee Person cardinality)
        to: #(employer Company cardinality)
        withAttributes: 'salary titel'
```

Here we have added cardinality as part of the role specifications, and have added that each association also has attributes salary and title. These attributes can be implemented as instance variables in the relation objects (associations).

In the underlying code, this creates WorksFor as a subclass of Relation. In this new class, we also store the role names, types and cardinalities of the relation. This infor-

mation is unique to each relation, and should therefore be stored in instance variables of the meta-class Relation class[3].

7.2 ACCESSING RELATIONS

We provide access to the relation at three different levels.

- There are the general access methods defined in the meta-class for Relation, which implements the functionality of adding and removing associations, querying, and selecting elements. As an example, the method index1: elem will return the set of objects related to elem with elem as index1.

- Next, we define customized access methods for the concrete relations, methods that draw upon the general methods of Relation class. Continuing with the above example, a method employee: elem is added to the WorksFor relation.

- Finally, we create access methods in the role types themselves. This allows us to find all the employers for a given person by sending the message employer to a person object.

In the following we will address adding and removal of associations, testing and querying at each of these three levels.

7.2.1 Adding associations

One necessary operation on relations is to be able to add new elements. The basic operation for adding a pair of elements is (exemplified by the WorksFor relation):

```
WorksFor insert: 'Jim Jones' and: 'Acme Products'
```

What we do is that create (and return a new association), and add it to the relation. It must be checked that elem1 and elem2 are of the appropriate types for this relation and that we respect the cardinality constraints.

By utilizing the role names, we provide a special access method for adding, which have the form:

```
WorksFor     add_employee: 'Jim Jones'
        and_employer: 'Acme Products'
```

The implementation is straight forward, and is done by compiling methods into the subclass (of Relation, in this case WorksFor) when the class is created.

Finally we can use the methods compiled into the role types:

---

[3] Instance-variables of meta-classes are a feature not often discussed in Smalltalk. They differ from class variables in that class variables are shared by the class, all its subclasses, and all their instances. Instance variables of meta-classes are only visible in the meta-class itself, and each class will have its own, just like each instance variable is unique to each object.

```
'Jim Jones' employer: 'Acme Products'
```

Here we have no syntactic indication that the WorksFor relation is used, only the role names (roles) are used here. This means that for a given class, it cannot play the same role in two different relations.

7.2.2 Removing associations

Just as we can add associations, there must be methods to remove them. This is done using the method:

```
WorksFor remove: elem1 and: elem2
```

The role names are not used to create a special remove method in the WorksFor relation (though a remove_employee:from_employer: method could be added). However, a remove method is provided directly in the participating objects:

```
'Jim Jones' remove_employer: 'Acme Products'
```

7.2.3 Membership testing

The basic operation for membership testing is:

```
WorksFor testMember: 'Jim Jones' and: 'Acme Products'
```

This operation works by creating a new relation object relating elem1 and elem2, and checking to see if this is already a member of the relation. This sort of testing is not done as often as querying and no special methods have been added.

7.2.4 Querying

In a binary relation, it is natural to want to find the objects to which a given object is related. We provide two types of querying, simple querying which returns the set of objects related to a given object, and association query, which return the set of associations containing an given object in a given role.

Simple querying returns the set of objects a given object is related to. There are two operations for doing this.

```
WorksFor atIndex1: elem1
```

returns the set of role2 objects that has elem1 as first object. The atIndex2: works similarly.

In our WorksFor example we would like to index the relation using the actual role names rather than the generic index1 and index2. We thus prefer a more readable syntax like:

```
WorksFor employee: 'Jim Jones'
```

which returns the set of employers which, has Jim as employee.

As the final layer, we provide a method in Person, which can return the set of companies related to Jim through a WorksFor relationship:

```
'Jim Jones' employer
```

Notice again that the Person cannot play the same role in two relations.

Association querying returns the set of associations that has a given object as its role.

```
<RelationName> relObjAtIndex1: elem1
```

Returns the set of relation-object that has elem1 as first role -relObjAtIndex2: works similarly. This type of query is used access the attributes of associations. In the WorksFor example we can use this to access salary and title.

Currently no second level access is implemented, but one can obtain the associations directly from the objects themselves.

```
'Jim Jones' employerAssociations
```

return all associations which has Jim Jones as employee.

The associations have methods that return the objects it associate. These can be accessed using the declared role names. Thus 'Jim Jones' employerAssociations first employer returns the first employer for Jim Jones'.

All querying methods return a set of related objects. This is useful in connection with Smalltalk's build-in enumeration methods, and gives some similarity to query languages.

```
'Acme Products' employee
        select: [:employee | employee zipCode = 'Acme Products' zipCode]
```

finds all the employees which has the same zip code as the company.

```
('Acme Products' employeeAssociations
  select: [:association| association salary > 35000])
  collect: [:association | association title]
```

first finds all the associations that associates an employee to Acme Products, which has a salary above 35000 (remember that salary is a variable on the associations, not the employees or employers). Then the titles from all these associations are collected. If the employees were interesting rather than their title, the last line would have read:

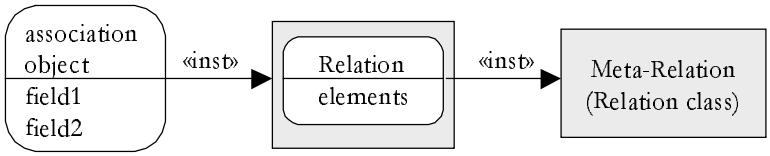
```
collect: [:association | association employee ]
```

7.3 IMPLEMENTATION

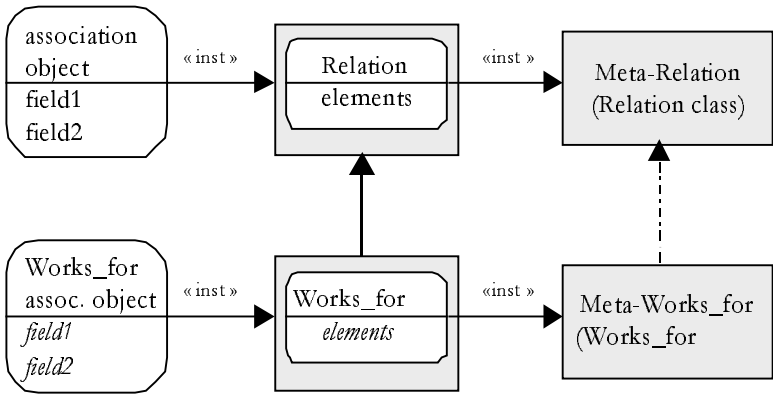
In this section we will examine how the relations have been implemented in Smalltalk. Meta-classes play a major role, so we will briefly introduce them in the context of the developed relation concept. Meta classes arise from the simple statement that every thing is an object – hence classes should be objects as well. Hence, a class A is an object, and should therefore be an instance of a class. This class is called the meta-class of A. This is again an object, hence there must be a meta-meta-class, which is an object, hence, … For our purpose it is sufficient to consider the class and the meta-class.

In our relation implementation, Relation have instances, which we throughout this sec-

tion has called association objects. The Relation class is an object. This object is what we refer to when we talk about *the relation*.



The above figure illustrates the object and class role of the class Relation. Classes are indicated in gray boxes, and objects in white, with rounded corners. An association object is an instance of (indicated by the label «inst») the Relation class, and each association object has two variables, field1 and field2. Relation is also an object, which is an instance of Meta-Relation[4]. The relation object contains all the elements, that is all the association objects that make up this relation.



In Smalltalk, each class has a unique meta-class, so no two classes are instances of the same meta-class. The inheritance hierarchy of the meta-classes follows that of the classes. This is easy to handle, as the meta-class is created automatically when the class is created. We use this property in our implementation when we create Relations. A Relation like “Works_for” is created by making a subclass of Relation. Works_for association objects will have two fields, because Works_for is a subclass of Relation, and the Works_for class-object will have its own elements, which are again the elements that make up the Works_for relation.

By creating Works_for as a subclass of Relation (shown using the bold arrow), the meta-class is created automatically. The effect is that association fields and elements are inherited (indicated using *italics*). The method named:relating:to: creates a new subclass when send to the Relation object as seen ealier:

```
Relation named: #WorksFor
        relating: #(employee Person)
        to: #(employer Company)
```

There is several ways to represent relations, but the initial representation chosen is that

[4] In Smalltalk terminology, this class is usually written “Relation class” rather then Meta-Relation. However, we find it easier to consistently refer to it as the meta-class it is.

we use a Set of relation-objects.

7.3.1 Implementation of special access methods

The general technique used to implement the special access methods is the possibility to add new methods to a class at run-time. When we have created the new relation (e.g. Works_for), we can add new methods to it by calling the compile: aString classified: aCategory method. The string must be a string representing a method in legal Smalltalk syntax; a method will then be added to Works_for. The methods added are really just access methods, which provides alternative names, e.g. employee rather than field1.

We have provided a division of labor, in that all actual work is done in methods in Relation and Meta-Relation. An alternative would have been to compile the actual methods into each relation class, but that seems to miss the point of reuse and sharing.

7.3.2 A few other loose ends

There are a number of methods, which should be defined for most Smalltalk objects, and which have nothing to do with relations.

- It is practical to be able to print both relation-objects and entire relations. Relation-objects can be printed as: "<obj1, obj2>''. This is quite simple because we will assume that both obj1 and obj2 are able to print themselves. To print an entire relation, we must take into considerations that it can be large, and should therefore not print all elements. The relation is considered a set of associations, and could be printed as: "RelationName: {<a1,b1> <a2,b2> ... }'', where we after some n (currently 20), will just print three dots and stop.

- In order to make the relations display right in the browser we need to redefine the definition method. This is a technical detail, which has to do with the fact that class definitions are not stored as text (as methods are), but are pretty printed on demand. Thus, we need relations to pretty print right.

8 ORACLE IMPLEMENTATION

We will examine how to implement the ideas of Rumbaugh in a relational database. We use Oracle 8 with its new object oriented features to see how close we can come to implement Rumbaugh's model.

8.1 CREATING CLASSES

The classes in the model maps directly to a table in a relational database. This is well known when we implement a data model in modern systems. In the new object-relational databases like Oracle 8, we would though create tables using object types instead of the traditional way of using simple datatypes for the table columns. By using object types we also get the possibility of adding instance methods for the Person class. The following statements creates the Person class

```
create or replace type Person_Type as object (
  Name varchar2(50)
  -- other attributes
  -- Person instance methods could be added here as well
 );
create table Person of Person_Type;
```

8.2 CREATING RELATIONS

When we implement a traditional data model in Oracle, we usually transform the relations depending on cardinalities. If it is a many-to-many relation, it is transformed to a table containing foreign keys as references to the tables representing the participating entities. A one-to-one or one-to-many relation could be implemented the same way though because of performance considerations they are usually implemented as foreign keys in the participating tables themselves.

The same strategy is generally used when implementing an object model in a relational database.

In this report, we have argued that relations are an important abstraction mechanism in the implementation as well so we will implement the relations in the model as tables in Oracle.

A table contains tuples of related values and it is well suited to implement a relation. An interesting question is whether a relation has identity. At present, we haven't researched this issue in detail but it affects how we should implement the relationships i.e. whether we should use an object type or not. Seen from the programmers standpoint there isn't much difference in how relations are accessed through SQL-statements but we assume it will give more possibilities in the procedural programming if we use object types. It is then possible to add methods to the relation object type and it is easier to refer to a relationship in the code. Another argument could be that we want to implement the relations at the same level as the classes.

We will therefore use an object type for the relationships and implement the relations in a manner similar to the implementation of classes.

Rumbaugh creates the Works_for relation with the syntax

```
RELATION Works_for (employee: Person, employer: Company)
```

In the Oracle implementation, this is equivalent to the statements

```
create or replace type Works_for_Type as object (
  Employee ref Person_Type,
  Employer ref Company_Type);
create table Works_for_Rel of Works_for_Type;
```

8.3 ACCESSING RELATIONS

To access the relations we need functionality to add and remove associations and also to query and select elements. The SQL language implementation in Oracle provides access to the relation through the SELECT, INSERT, UPDATE and DELETE statements. One could stick with these statements, as they are indeed very flexible to use for accessing a relation.

As another approach, we will present an implementation of access methods that is close to the examples of Rumbaugh. In this implementation, we meet some of the limitations of the current release of Oracle 8. Though Oracle 8 supports objects with attributes and methods, the methods cannot use statements, which modifies the database tables such as the INSERT, UPDATE and DELETE statements. In general the methods are not suited for class methods but are rather intended for instance methods.

We will instead follow a good practice using Oracle packages to encapsulate functionality for each relation. Packages provides some powerful object-oriented features as described in [5]. For each relation we create a package with the same name as the relation itself. It will then look as if the package methods (procedures and functions) really belong to the relation though it is only by naming convention it is achieved.

When using the package solution we should consider if we only want access to the relation through the package methods and in that case establish access rights to secure it.

8.3.1 Adding associations

Rumbaugh use a syntax for adding associations like in the example

```
Works_for.Add(Allan, RAMBØLL)
```

If Allan and RAMBØLL are known objects at runtime, the equivalent INSERT statement in Oracle is

```
insert into Works_for_Rel values (Works_for_Type(Allan, RAMBØLL));
```

To select the objects in the INSERT statement it would instead look like

```
insert into Works_for_Rel
select ref(p), ref(c) from Person p, Company c
where p.Name = 'Allan' and c.Name = 'RAMBØLL';
```

We encapsulate the INSERT statement in an Add method by creating the Works_for package like

```
create or replace package Works_for as
  procedure Add(AnEmployee ref Person_Type,
    AnEmployer ref Company_Type);
  -- Other Works_for methods
end Works_for;
create or replace package body Works_for as
  procedure Add(AnEmployee ref Person_Type,
    AnEmployer ref Company_Type)
  is
  begin
    insert into Works_for_Rel
    values (Works_for_Type(AnEmployee,AnEmployer));
  end Add;
-- Other code for Works_for methods
end Works_for;
```

We could then use the following code to add the relationship that Allan works for RAMBØLL.

```
declare
  Allan ref Person_Type;
  RAMBØLL ref Company_Type;
begin
  select ref(p) into Allan from Person p where p.Name = 'Allan';
  select ref(c) into RAMBØLL from Customer c where c.Name = 'RAMBØLL';
  Works_for.Add(Allan, RAMBØLL);
end;
```

8.3.2 Removing associations

To remove associations we use the DELETE statement

```
delete Works_for_Rel w where w.Employee = Allan and w.Employer = RAMBØLL;
```

or

```
delete Works_for_Rel w
where w.Employee.Name = 'Allan' and w.Employer.Name = 'RAMBØLL';
```

depending on whether Allan and RAMBØLL are known objects at runtime.

The Remove method in the Works_for package is straightforward

```
create or replace package body Works_for as
  procedure Remove(AnEmployee ref Person_Type,
    AnEmployer ref Company_Type)
  is
  begin
    delete Works_for_Rel w where w.Employee = AnEmployee
    and w.Employer = AnEmployer;
  end Remove;
-- Other code for Works_for methods
end Works_for;
```

The Remove method is called the same way as the Add method.

8.3.3 Membership testing

To test for membership we use the SELECT statement. The result can be given as Boolean, or we can handle a found or not found result, which is slightly simpler to code.

The expression to test whether Allan works for Aalborg University

```
Work_for.Test_member(Allan, Aalborg University)
```

can be written as a simple SELECT statement

```
select * from Works_for_Rel w
where w.Employee.Name = 'Allan'
and w.Employer.Name = 'Aalborg University';
```

The code then needs to check for a NOTFOUND exception.

We could consider to write a Test_member method in the Works_for_Type. Since the method is a method for the relation (class method) in contrast to being a method for the relationships (instance method) the correct place for the method is in the relations package. We then implement the Test_member method as

```
create or replace package body Works_for as
  function Test_member(AnEmployee ref Person_Type,
    AnEmployer ref Company_Type) return boolean
  is
    result boolean;
    dummy number;
    cursor TestCur is
    select 1 from Works_for_Rel w
    where w.Employee = AnEmployee and w.Employer = AnEmployer;


  begin
    open TestCur;
    fetch TestCur into dummy;
    if TestCur%FOUND then
      result := TRUE;
    else
      result := FALSE;
    end if;
    close TestCur;
    return result;
  end Test_member;
-- Other code for Works_for methods
end Works_for;
```

An example of testing whether Allan works for Aalborg University using the Test_member method could be written as

```
declare
  Allan ref Person_Type;
  Aalborg ref Company_Type;
begin
  select ref(p) into Allan from Person p where p.Name = 'Allan';
  select ref(c) into Aalborg from Company c where c.Name = 'Aalborg University';
  if Works_for.Test_member(Allan, Aalborg) then
    -- do something
  end if;
end;
```

8.3.4 Querying

The SQL language is very flexible for querying. After all SQL was designed to be a query language and to query for relationship we can use the SELECT statement in many different ways. It will be too overwhelming to show all the variations in this context so we only show the simple querying here where we query the set of objects which are related to a given object based on a given role.

To find the employees who works for RAMBØLL we use the SELECT statement

```
select Employee from Works_for_Rel w
where w.Employer.Name = 'RAMBØLL';
```

We will provide a getEmployees method, which encapsulates this query and can be called like

```
Works_for.getEmployees(RAMBØLL)
```

The method returns a set of objects i.e. Persons and we need to create a type to express it

```
create or replace type Person_Set as table of ref Person_Type;
```

We can then implement the method in the relation package

```
create or replace package body Works_for as
  function getEmployees(AnEmployer ref Company_Type) return Person_Set
  is
    Employees Person_Set;
    n binary_integer := 0;
  begin
    Employees := Person_Set();
    for Emp in (select Employer from Works_for_Rel w where w.Employer =
AnEmployer)
    loop
      n := n + 1;
      Employees.Extend;
      Employees(n) := Emp.Employee;
    end loop;
    return Employees;
  end getEmployees;
-- Other code for Works_for methods
end Works_for;
```

To call the getEmployees method we use code like

```
declare
  EmployeesInRAMBØLL Person_Set;
  RAMBØLL ref Company_Type;
begin
  select ref(c) into RAMBØLL from Company c where c.Name = 'RAMBØLL';
  EmployeesInRAMBØLL := Works_for.getEmployees(RAMBØLL);
end;
```

A similar method getEmployers would also be implemented.

Finally we will provide methods in the Person and Company object types which will return the set of related objects through the Works_for relation. The methods will use the previously implemented query methods. In the Company_Type we will implement an Employee method which uses the getEmployees method.

```
create or replace type Company_Type as object (
  Name varchar2(50),
  -- other attributes
  member function Employee return Person_Set,
  -- Other Company instance methods could be added here as well
  pragma restrict_references(Employee, WNDS, WNPS)
 );
create or replace type body Company_Type as
  member function Employee return Person_Set is
    result Person_Set;
    ref_self ref Company_Type;
  begin
    select ref(s) into ref_self from Company s where value(s) = self;
    result := Works_for.getEmployees(ref_self);
    return result;
  end Employee;
);
```

Notice the pragma informs that the method will have no side effects on the database tables.

We can call the access method in an example like

```
declare
  EmployeesInRAMBØLL Person_Set;
  RAMBØLL Company_Type;
begin
  select value(c) into RAMBØLL from Company c where c.Name = 'RAMBØLL';
  EmployeesInRAMBØLL := RAMBØLL.Employee();
end;
```

Notice however that the access method cannot be used in a SELECT statement like

```
select c.Name, c.Employee() from Company c;
```

because the method returns a set of objects. We have to code this kind of code in a different way, and we really would like some extra support for this in Oracle.

9 REFERENCES

[1] A. V. Velho and R. Carapuça. *Attribute: A Semantic and Seamless Construct*. http://albertina.inesc.pt/ESW/documents/som-tol.

[2] J. Rumbaugh. Relations as Semantic Constructs in an Object Oriented Language, *Proceedings of OOPSLA'87*. Pages 466-481.

[3] A. V. Shan, J. Rumbaugh, J. H. Hamel, and R. A. Borsari. DSM: An Object-Relationship Modeling Language. *Proceedings of OOPSLA'89*. Pages 191-202.

[4] P. Coad, *Object-Oriented Patterns*, 1992, Communications of the ACM, September 1992, pp. 152-159

[5] Oracle Unleashed, Sams Publishing, chapter 18. http://technet/oracle.com/odp/public/connect/sams/html/oun.htm

[6] S. T. March and S. Rho. A Semantic Object-Oriented Data Access System. http://www.misrc.umn.edu/wpaper/wp96-05.htm

[7] J. Olsson, K. H. Nielsen, K. Østerbye, A. R. Lassen. Objektorienteret Analyse og Design af udvalgte dele af STADS. COT/4-03. (in Danish)