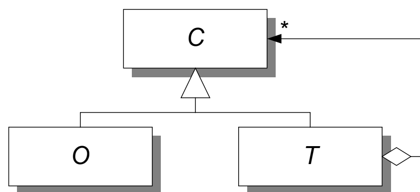
PERSISTENT STORAGE OF OO-MODELS IN RELATIONAL DATABASES

COT/4-02-V1.5



Centre for Object Technology

Revision history:	V1.0	12-09-97	First version
	V1.1	15-09-97	First public version
	V1.2	02-11-97	Second public version
	V1.3	28-11-97	Revised version based on comments
	V1.4	13-03-98	Final version using shared COT layout
	V1.5	22-04-98	Final, final version using new shared COT layout

Author: Michael Thomsen, DAIMI. Email: `miksen@daimi.aau.dk`
Supervisor: Jørgen Lindskov Knudsen, DAIMI

Status: Final

Publication: Public

Summary:

This document describes central problems arising when trying to achieve persistent storage of object models in relational databases

© Copyright 1998

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 Center 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 5

2. A QUICK TOUR OF RELATIONAL DATABASES 7

2.1 TABLES 7
2.2 KEYS AND INDEXES 7
2.3 NORMAL FORMS 7
2.4 SQL 8

3. PERSISTENT STORAGE IN GENERAL 9

4. STORING OBJECTS IN RELATIONAL DATABASES 10

4.1 SO WHAT DO WE WANT? 10

5. TRANSFORMING AN OBJECT MODEL 12

5.1 THE CLASS 13
5.2 GENERALISATION HIERARCHIES 13
5.2.1 Separate tables mapping 13
5.2.2 Leaf-classes only mapping 14
5.2.3 Single table mapping 15
5.3 ASSOCIATION 15
5.3.1 Many-to-many associations 16
5.3.2 Many-to-one associations 16
5.3.3 One-to-one associations 17
5.4 AGGREGATION 18
5.5 SUMMARY 18

6. TACKLING EXISTING DATABASES 20

6.1 INCOMPATIBILITIES 20
6.2 INEFFICIENCIES 21
6.3 SUMMARY 22

7. HANDLING OUR DATABASE OPERATIONS 23

7.1 HANDLING FETCHES 24
7.1.1 Where to fetch what 24
7.1.2 Uniqueing 24
7.1.3 How to handle relationships 26
7.2 HANDLING UPDATE 26
7.3 HANDLING SAVE 26
7.4 HANDLING DELETE 27
7.5 SEARCHING FOR OBJECTS 28
7.6 THE DATA MAPPING 29

8. SUMMARY 30

9. BIBLIOGRAPHY 31

TABLE OF FIGURES

FIGURE 1. THE BASIC DATABASE OPERATIONS....11
FIGURE 2. SMALL UNIVERSITY MODEL....12
FIGURE 3. UNIVERSITY....13
FIGURE 4. UNIVERSITY DEPARTMENT....13
FIGURE 5. PERSON....14
FIGURE 6. STUDENT....14
FIGURE 7. DEPARTMENTHEAD....14
FIGURE 8. STUDENT....14
FIGURE 9. DEPARTMENTHEAD....15
FIGURE 10. PERSON....15
FIGURE 11. UNIVERSITYDEPARTMENT – STUDENT ASSOCIATION....16
FIGURE 12. UNIVERSITY DEPARTMENT....17
FIGURE 13. UNIVERSITY DEPARTMENT AND DEPARTMENT HEAD....17
FIGURE 14. MAPPING GENERALISATION HIERARCHIES....18
FIGURE 15. MAPPING ASSOCIATIONS....19
FIGURE 16. INCOMPATIBILITY EXAMPLE 1.....20
FIGURE 17. INCOMPATIBILITY EXAMPLE 2.....21
FIGURE 18 THE BASIC DATABASE OPERATIONS....23
FIGURE 19. THE UNIQUEING PROBLEM....24
FIGURE 20. DATA MAPPING REQUIREMENTS....29

1. INTRODUCTION

Since the first OO language – Simula in 1967 – object oriented languages have become widely available and they have furthermore been accompanied by object oriented analysis and design techniques. These techniques have reached a level of maturity, and are widely used in the industry.

There is still one part of the object oriented paradigm that seems not to have been fully embraced by the industry yet though, namely object oriented storage mechanisms for persistent storage of runtime objects. Often relational databases seem to be favoured.

There are many reasons for this. Some are technical and theoretical problems, but probably the most important argument is that there are thousands if not millions of systems in operation at the current, which all are based on relational databases, and shifting all of these to a new database technology is at least for the time being too costly both in money and risk of failure.

Recognising that relational databases are necessary even when developing new applications and systems using object oriented techniques, this report will discuss the relationship between object models and relational databases. This includes two related issues: We must investigate how we can provide persistent storage of our object models in the relational databases, and how we can provide OO access to data stored in relational databases.

There have generally been two approaches to this problem: A bottom-up approach and a top-down approach. The bottom-up approach focuses on constructing an object model that reflects the data in an existing database. The top-down approach, on the other hand, takes the starting point in the object model constructed using the usual modelling techniques and from this designs a new database with a number of tables that the objects can be stored in.

In this report we will try to take a mixed approach, as we believe that both issues – the existing database and the object model – should be taken into regard. The existing databases are important, as often we would like to keep these instead of having to create new ones. Constructing the object model using modelling techniques and not just as direct mapping of some database is also important, as the object model ideally should reflect the part of the world that the computer system will be representing, and not some database design [MMN93, pp. 289].

We realise that of course "we can not have the cake and eat it too", in the sense that taking an existing database and an object model where both have been constructed without paying attention to the other at all will probably be problematic. We might not be able to make the integration at all, and if we should make it, then the transformation will probably have a lousy performance. Therefore we will in the report point out where this ideal approach will not be reasonable.

The outline of the report will be as follows: We start off with a small description of some of the main concepts of relational databases, then we discuss general issues in persistence and then continue on to discussing in general terms what we want from our integration to relational databases.

In the main part of the report we will discuss the integration with relational databases, look into what the problems with this integration are, and investigate how to get around them. Specifically we will look at how to save the state of an object model in tables in a relational database and discuss how it will influence us if the state is to be saved to an existing database. Finally we will look at four central database operations and which problems we will face when implementing these.

Finally it should be noted that some of the more complex aspects of relational databases, such as triggers and stored procedures, we consider to be out of scope in this report.

2. A QUICK TOUR OF RELATIONAL DATABASES

This chapter will give a quick introduction to some of the most important aspects of relational databases. We by far do not cover all there is to be said about relational databases, but only touch upon those issues that are relevant for further discussion of our problem.

Readers who are familiar with relational databases can skip this section, and continue to section 3.

2.1 TABLES

A relational database consists of a number of tables. Each table has a specific number of columns denoted attributes, and a (virtually) arbitrary number of rows called tuples (or records). The intersections of a row and a column are called fields, and these fields contain the data of the table. All fields within a column have the same type, which is of a single or atomic datatype. The collective definition of all the tables constitutes the conceptual scheme of the database or simply its scheme.

2.2 KEYS AND INDEXES

Each table in the relational database always contains one or more candidate key, one primary key, possibly a number of foreign keys and maybe a number of indexes. A candidate key consists of a minimal number of attributes that uniquely identify each and every row in the table, and can thereby be used to locate the rows one by one. For each table a candidate key is chosen to be the primary key.

A foreign key is a set of attributes that is a primary key in another table, and these are usually used to join tuples together across tables. Finally indexes are attributes that are often used for searching and therefore should be optimised for faster access by the database system.

2.3 NORMAL FORMS

Tables can be said to be in so called normal form. The purpose of normal forms is to avoid redundancy and inconsistencies when updating the table contents. First normal form only specifies that each attribute value is an atomic value. A value is said to be atomic if it is an indivisible unit. For example a name attribute consisting of both surname and first name would not have atomic fields.

In second normal form the table must furthermore satisfy that each attribute is either part of the primary key, or that the attribute depends on the full primary key. Should an attribute not part of the primary key only depend on part of the primary key then the table must be split up into several tables. An example of a violation could be a car table having the attributes make, model, weight and logo with primary key (make, model). Here the logo of the car does not depend on both make and model, but only on the make, and the table thereby violates second normal form.

Finally third normal form adds an additional restriction. Assume that a field is nonprime, that is a field not in any candidate key. Then this field may not depend transitively on any

candidate key, where we say that an attribute C is transitively dependent of an attribute A, if C is dependent on an attribute B that is dependent on the attribute A.

As it should have been hinted by now, bringing a database into normal form can drastically increase the number of tables, but nevertheless reduces the number of update inconsistencies. For a more thorough discussion of normal forms see [BEL+91] or [KoSi91].

2.4 SQL

Queries in relational databases are normally done using the query and creation language SQL (Structured Query Language). Searches are normally in the form of `SELECT` statements that basically tells the database which attributes to fetch from which tables, under the condition that they meet some specifying criteria.

3. PERSISTENT STORAGE IN GENERAL

Normally data found in memory during a program execution will only be available for that one execution. As soon at the execution ends the data will be gone. This data is so called *transient* data.

In many situations we need some mechanism for making these data objects available for later executions. This is usually done by saving the state of the objects to secondary storage such as a harddisk or a tape. In object oriented terms we will say that an object whose state has been saved in such a way that it later can be used by another program execution, has been made *persistent*, and we will refer to the mechanism that makes these objects persistent as *persistence* [Mjø94].

Often there is another major reason for saving the state of the runtime objects to secondary storage: To support sharing of data between several concurrently running executions. This sharing can done using the same mechanism as for persistent storage with some functionality added for controlling simultaneous access to the same data.

Furthermore we say that a persistent storage mechanism can be *orthogonal* and *transparent*. By orthogonal persistence we mean that the ability to make an object persistent is independent of the type of the object. For example we cannot require that the class of the object is a subclass of a designated class such as `PersistentClass`, or that the object has certain methods that we use to implement the persistence.

By transparent we mean that the persistence is not "visible" to the application programmer. For example we do not require that he should assign objects to a `saveNow` procedure in order to make them persistent, or that he should respond to an `objectDataHasChanged` event when the state of the object changes on the disk.

When designing a persistent storage mechanism we have to consider if we wish to offer one of both of these two properties. Seldom though can we offer them in their purest form, as technical issues can limit us to making them only almost orthogonal or almost transparent. For example the persistent storage in the Mjølner Beta System [Mjø94] is orthogonal and almost transparent in the sense that once it has been told what to make persistent it will be transparent.

4. STORING OBJECTS IN RELATIONAL DATABASES

In the previous chapter we discussed what we meant by persistent storage. In this chapter we will look into how this persistence of our objects can be achieved using relational databases. This is by no means an easy task.

The main problem in achieving this persistence using relational databases is caused by the fundamental differences between the two levels of data description. Where the relations in the relational databases can only contain atomic values, there are no restrictions on the possible attributes of the objects in the OO-model. Object oriented languages have structural constructs that are not found in the relational data model, such as generalisation, association and aggregation. We therefore need some way of mapping the state of the objects – which is in a high level of abstraction – into the tables which are in a lower level of abstraction.

4.1 SO WHAT DO WE WANT?

Before describing in detail what the problems are we should describe what it really is that we want from our persistent storage mechanism, in the light of the fact that we are going to implement it using a relational database.

First of all it would be nice if our solution could offer orthogonal persistence. Restricting persistence to only some classes can render it useless if the application should in fact use any classes not part of those. Requirements such as objects that are to be made persistent should be defined by classes being subclasses of some persistent superclass, is a little better, but can also be critical – especially in languages not having multiple inheritance. All in all it is worth looking into whether it is possible to achieve orthogonal persistence, or a least virtually orthogonal persistence.

Transparency on the other hand is impossible to achieve in a sensible way: As we mentioned earlier, saving data to secondary storage is often not only used to make objects persistent but also to share data between (concurrent) program executions. If this is the case there are a number of problems that needs to be addressed to achieve this sharing, which makes it unpractical if not impossible to achieve transparency.

This would for example requiring that the system should now when to place locks and when an object is ready to be saved to be made available for the other execution. As this of course is impossible, or at least impractical as it would end up locking/saving far too often, we in general will not be able to achieve a meaningful transparency when have shared data.

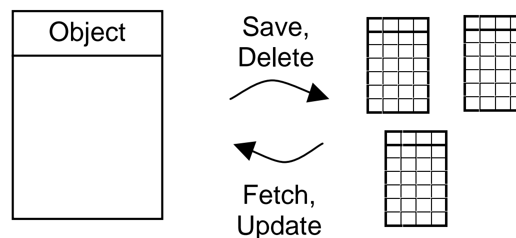
In terms of functionality there are quite a few operations which one could wish to have, in a persistent storage mechanism. Some examples are:

- Fetch an object from the database
- Search for an object or a number of objects and fetch only those
- Update the state of an already fetched object
- Save the state of an already fetched object

- Add an object to the database
- Delete an object from the database

Since some of these operations can be simulated by the others, we do not need them all. For example fetching a single object can be compared to searching for a single object, and saving an object can be applied both to those not in the database and those already in the database. As a starting point we therefore restrict ourselves to four operations: *save*, *delete*, *update* and *fetch* as illustrated in Figure 1 below.

Figure 1. The basic database operations



Save: The save operation takes as input a single object. If the object is already in the database, the data in the database corresponding to the object will be changed according to the values in the object. If the object is not in the database it is simply added to the database.

Delete: The delete operation takes as input a single object that has previously been fetched from the database, and deletes the data in the database corresponding to the object.

Fetch: The fetch operation takes as input a class and returns a list of all those instances of that class that are currently found in the database. To restrict the number of returned objects search criteria can be added.

Update: The update operation takes as input a single object that has previously been fetched from the database. It then compares the values in the object with the values in the database, and if the values in the database have been changed they will be fetched and the values in the object changed.

In the next couple of chapters we will look into the problems involved in implementing these four basic functions. We start by showing that going from an object model, it is in fact possible to represent the classes and the relations between these in tables in a relational database. After this we consider how our design of the object model is influenced if we have to save it in an existing relational database, and not construct a new one for the purpose.

We then return to our four database operations, and look at which problems we can encounter when implementing them and how they possibly can be solved.

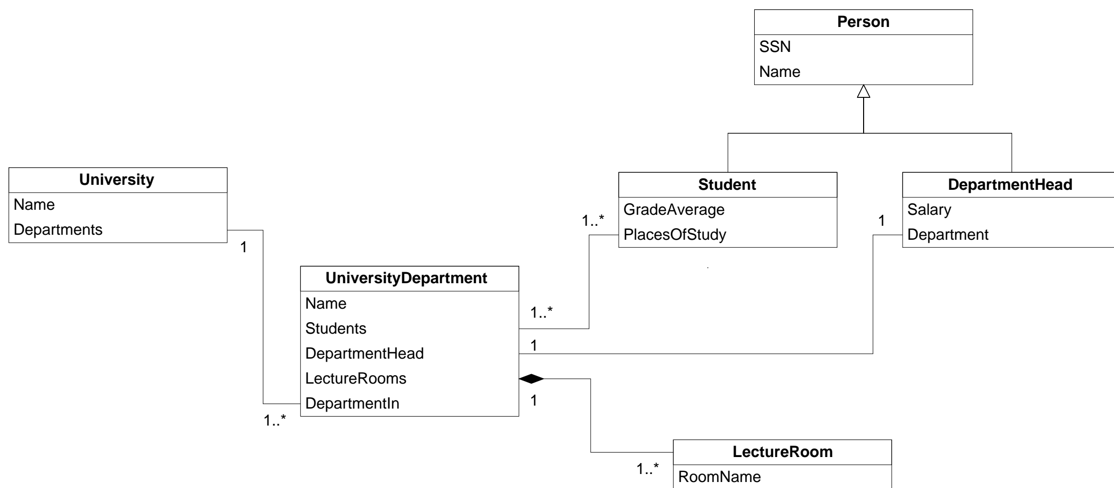
5. TRANSFORMING AN OBJECT MODEL

When mapping an object model to a relational database two situations can exist: We currently have no existing database for the system and we can therefore create new tables, or we already have a database, and we therefor have to map the model to these existing tables.

In this chapter we will look at which possibilities we have if we are doing the mapping to a "fresh" database. This gives us much freedom and allows us to chose the solution that is most optimal in terms of speed and space. In the next chapter we then look into how our choices are influenced if an existing database has to be taken into account.

The transformation of objects into tables is concerned with four constructs: the class, the subclass (specialisation), the association and the aggregate. In turn we will give examples of how to transform these. The examples will all be taken from the small (somewhat artificial) OO-model modelling a university. The model is shown below in Figure 2 using UML Class-diagram notation and the mapping suggestions are as seen in [BEL+91], though with minor modifications.

Figure 2. Small university model



The example OO-model contains the following: The superclass `Person` modelling general aspects of a person, such as his/hers SSN (Social Security Number). The subclass `Student` modelling a person that is also a student. The subclass `DepartmentHead` modelling a person that is also the head of a university department. The class `University` that models overall aspects of a university. The class `UniversityDepartment` that models a department of a university and finally the class `LectureRoom` the models a room for giving lectures at a university. As seen in the model one university is associated to one or more university departments. One or more university departments associate one or more students, one university department associates one department head and one university department aggregates one or more lecture rooms.

5.1 THE CLASS

The main choice when mapping the class is whether there should be one table per class or several tables per class. As a general rule Blaha et al [BEL+91] write that one should construct one table per class, but this is not always the case. Should the mapping to one table violate third normal form, the table will have to be split up vertically into two or more tables. If performance aspects are critical and if the number of records be large, then the table can by split up horizontally into a number of tables representing groups of tuples.

Furthermore a primary key must identified. Should the class already have an attribute or a number of attributes that can be used as a primary key they will be used, but if not, one must be introduced. Alternatively the OID (Object IDentifier) can be used, if it is possible to introduce an attribute in the table to save this in.

Taking the university class as an example we do not have an attribute that can be considered unique (as theoretically several universities can have the same name), so we will have to introduce one for our primary key. The resulting table for the university class – which conforms to third normal form – is shown below in Figure 3 and the result for the university department class mapped using the same rules is shown in Figure 4.

Figure 3. University

Entity Name	**Required?**	**Type**
UniversityId	Yes	Integer
Name	Yes	Text 100
Primary key: UniversityId		

Figure 4. University department

Entity Name	**Required?**	**Type**
DeptId	Yes	Integer
Name	Yes	Text 100
Primary key: DeptId		

5.2 GENERALISATION HIERARCHIES

Just as with the class there are several possibilities when mapping superclasses and subclasses. Blaha et al presents three possibilities, which all haves advantages and disadvantages depending on the actual hierarchy. The mappings we describe here only take single inheritance into consideration.

5.2.1 Separate tables mapping

The first mapping maps each superclass and subclass to separate tables. This mapping is done using the standard mapping for classes, with two modifications: The primary key in the topmost superclass is also used as a primary key in the tables representing the subclasses, and the table representing the topmost class gets an additional attribute denoting which subclass is being represented. Using this mapping on the generalisation in our example model we get the following tables:

Figure 5. Person

Entity Name	Required?	Type
SSN	Yes	Text 10
Name	Yes	Text 100
PersonType	Yes	Text 20
Primary key: SSN		

Figure 6. Student

Entity Name	Required?	Type
SSN	Yes	Text 10
GradeAverage	No	Real
Primary key: SSN		

Figure 7. DepartmentHead

Entity Name	Required?	Type
SSN	Yes	Text 10
Salary	No	Integer
Primary key: SSN		

This mapping is useful when there are many subclasses on each level, but can result in a too large number of tables if the super/sub-class hierarchy becomes very high. Furthermore this mapping can be expensive when the attributes of subclasses have to be found, as several tables have to be consulted.

5.2.2 Leaf-classes only mapping

Where the previous mapping had one table per class, this mapping results in quite fewer tables. This is achieved by only creating tables for the subclasses that are at the very bottom of the hierarchy. These tables receive all the attributes of the bottom class, but besides this also receive all the attributes from any class met going up the tree from the bottom class to the topmost superclass.

In [BEL+91] an attribute indicating which class type the record contains is not added. They suggest that we can determine which class the record represents by testing which attributes that are null, and from this information deduce the type of the class.

This is not efficient and sometimes even not possible though. If for example the subclasses only have attributes that are null-able as in the case of the student and department head subclasses, we cannot see whether a record in the student table is a `Student` with the `GradeAverage` field not filled in or just a `Person`. If instances of the superclass person can occur we therefore have to add the attribute denoting the type. Without this addition the result in our example becomes:

Figure 8. Student

Entity Name	Required?	Type
SSN	Yes	Text 10
Name	Yes	Text 100
GradeAverage	No	Real
Primary key: SSN		

Figure 9. DepartmentHead

Entity Name	**Required?**	**Type**
SSN	Yes	Text 10
Name	Yes	Text 100
Salary	No	Integer
Primary key: SSN		

This mapping is useful when the superclass has few attributes and the subclass has many attributes, as the impact of putting the superclass attributes in every subclass table will not be very large. In can introduce inconsistencies though, as uniqueness of fields cannot be checked across tables. An example of this could be a database having both a student and a professor with the same SSN, in each of their tables. Furthermore any class not being the bottom subclass, will be mapped to a table that is "too large". This can result in a waste of space in some databases, and might reduce the efficiency.

5.2.3 Single table mapping

Where the philosophy of the second mapping was to bring attributes from superclasses "down" into the subclass level, the third mapping does the opposite. Instead of a number of tables for each subclass, we construct only one table having all the attributes of any class in the super/sub-class tree. This results in the following table:

Figure 10. Person

Entity Name	**Required?**	**Type**
SSN	Yes	Text 10
PersonType	Yes	Text 20
Name	Yes	Text 100
GradeAverage	No	Real
Salary	No	Integer
Primary key: SSN		

This mapping is useful when the hierarchy is not too wide. If the hierarchy furthermore is high, the lookup of attributes in subclasses low in the hierarchy is much faster than with the first mapping. The disadvantages are that it might result in a waste of space in some databases that reserve space for all attributes, that .

The disadvantages are of course that the table can become quite large. This can result in wasted space, in databases that do not optimise the space use of "half-empty" records.

5.3 ASSOCIATION

When mapping associations between classes, we can choose between introducing a new table mapping the association, or just introduce foreign key attributes that refer to the classes that are associated. Which strategy to choose depends on the multiplicity of the association.

5.3.1 Many-to-many associations

When mapping many-to-many associations we have to introduce a new table, to conform to the rules of third normal form. Take as an example the association in the above example model between the classes UniversityDepartment and Student. If we mapped this association by adding an attribute in the UniversityDepartment and Student tables which would refer to the primary key of the associated table, then our primary key would no longer be valid. This is easily seen as we could now have several records having only different values in the new association field, and thereby having the same value in the DeptId attribute, which is our primary key.

Making the association attribute part of the primary key could of course solve this problem, but at the same time produces a new problem. We will now violate second normal form, as some of the non-primary attributes no longer will depend on the full primary key.

We therefore always use a new table when mapping many-to-many associations. This table simply contains the attributes forming the two primary keys of the two associated tables. These are all required attributes, as associations only can be between existing objects. For the primary key we use the joint key of both attributes, as we can have records with the same values in either of the two attributes. The result of mapping the above mentioned association results in the following association table and the earlier shown UniversityDepartment (Figure 4) and Student (Figure 6) tables:

Figure 11. UniversityDepartment – Student association

Entity Name	**Required?**	**Type**
DeptId	Yes	Integer
SSN	Yes	Text 10
Primary key: (DeptId, SSN)		

5.3.2 Many-to-one associations

The mapping of many-to-one associations can of course be done using the technique for many-to-many tables discussed above, but this is no longer the only alternative, as the association table is no longer a requirement. Since one of the classes has a one multiplicity, we can represent the reference to this class by adding an attribute containing its primary key, to the table with the many multiplicity. This respects third normal form, and does not destroy the primary key as we never add multiple entries in the many table with only different values in the association field – this is easily seen as each distinct record in the many table is associated to only *one* other class. In our example model the association between university and university department changes the university department table from Figure 4 to:

Figure 12. University department

Entity Name	Required?	Type
DeptId	Yes	Integer
Name	Yes	Text 100
UniversityId	Yes	Integer
Primary key: DeptId		

The advantage of this mapping compared to the mapping of many-to-many with an extra table is that access to the data becomes faster as only two tables need to be consulted and that the overall number of tables is lower. There are also disadvantages though. The introduction of the association attribute in one table moves the association into one of the classes where it conceptually belongs in neither but as a separate entity. Furthermore practice has shown that it is difficult to get association multiplicity's correct on the first model iterations which speaks in favour of introducing a separate association table so changes to many-to-many are trivial. We therefore recommend adding the association table when very fast access is not critical.

5.3.3 One-to-one associations

One-to-one associations can of course be mapped in the same manner as with many-to-many and one-to-many associations, but besides this they can be mapped in a third way. Since there will only be one object on both sides of the association, we can collapse not only the association table into one of the object tables, but collapse *both* of the associated classes *and* the association table into one table. This mapping used on the association between university department and department head results in the table shown in Figure 13 below.

Figure 13. University department and department head

Entity Name	Required?	Type
DeptId	Yes	Integer
Name	Yes	Text 100
SSN	Yes	Text 10
Salary	No	Integer
Primary key: DeptId		

One has to careful though, as this transformation will often violate third normal form. The advantages in this further collapse of tables are the same as before: Faster lookup and lower overall number of tables, and the disadvantages are also the same: It is "conceptually wrong" and a change of multiplication becomes difficult. Furthermore this mapping can not handle cyclic associations of the form where A associates B, B associates C and C associates A. We therefore again recommend that one-to-one associations be transformed with the extra associations table as with many-to-many associations.

5.4 AGGREGATION

In OO-models there is a distinction between association and aggregation, but when mapping these two constructs the same methods apply. In Blaha et al, all that is said about aggregations is that they can be mapped in the same ways as associations.

This is the case as aggregation can be viewed as a special case of one-to-many association. It should probably be investigated further is this really is the case in reality, or if it only holds in theory, and special care has to be employed when mapping these.

5.5 SUMMARY

We presented a number of mappings for the four basic constructs: the class, the subclass (specialisation), the association and the aggregate.

For the class we recommended that a "one table for one class" approach was used, but also mentioned that splitting vertically or horizontally could be done were performance aspects were critical. Finally we noted that a primary key for each class should be identified or introduced.

For mapping generalisation hierarchies we introduced three mappings that could be used, and which had each their advantages and disadvantages. The three mappings were "separate tables mapping" which introduced a table per class or subclass in the hierarchy, the "leaf-classes only mapping" which only had tables for the classes at the very bottom of the hierarchy and the "Single table mapping" which had only one table for all the classes. The advantages and disadvantages of each of these are summarised in the table below.

Figure 14. Mapping generalisation hierarchies

Implemented using	Advantages	Disadvantages
Separate tables mapping	Good for wide hierarchies Fetches of "father"-classes are fast	Fetches of "leaf" classes are slow High hierarchies result in a large number of classes
Leaf-classes only mapping	Fetches of "leaf" classes are fast	Fetches of "father"-classes are slower Can result in uniqueness problems Possible space loss
Single table mapping	Fast for small hierarchies	Can result in uniqueness problems Possible space loss

For associations we also introduced a number of possibilities. First of all it is always possible to map the association using a separate table holding the primary keys of the related classes. When mapping one-to-many and one-to-one associations it was furthermore possible to introduce an attribute with foreign keys of the associated class in the class with the many multiplicity. Finally in the case of one-to-one association we can in some cases use just one table for *both* of the associated classes *and* the association class. The advantages and disadvantages of these three possibilities are shown in the table below.

Figure 15. Mapping associations

Implemented using	Advantages	Disadvantages
Separate table	Handles all types of association Changes of multiplication is possible	Can be slower than the other methods
Association attribute	Less joins needed	Only possible for many-to-one and one-to-one associations Less general
Shared table	No joins needed	Only possible for one-to-one associations Less general Does not handle cyclic associations Often violates third normal form

Finally we concluded by shortly saying that aggregation can be handled by viewing it as a special case of one-to-many association, but at the same time noting that this probably should be investigated further.

6. TACKLING EXISTING DATABASES

We have now shown that if we start out with an object model, we can construct a database scheme that reflects the same data and the same structure as in the object model. This makes it possible to save the state of our object model as we were hoping for. Often though we will not be able to use the mapping rules in this direct manner. The reason for this is simply that this method assumes that the relational database is to be constructed from scratch. As one of the main reasons for using relational databases in the first place, is a strong wish to keep existing databases, tables and data, this is often not possible.

Furthermore we work under the assumption that these existing databases can not easily be changed, even at a detail level. This could be the case where several systems access the same database, and only one of the systems is to be replaced. I such a case a change to the database could result in a need for changing all the applications accessing it, which often is not feasible.

These two observations complicates things and we will have to deal with a number of problems. We will divide these problems into two categories: Those that make the object model impossible to use (denoted incompatibilities), and those that make the object model ineffective to use (denoted inefficiencies).

6.1 INCOMPATIBILITIES

At least some incompatibilities are to be expected when connecting to an existing database according to Broløs and Pilgaard [BrPi95]. Unfortunately they do not mention exactly which connections they had trouble making.

It seems possible though that one incompatibility between an object model and an existing database, could be that the object model contains attributes for which there are no data in the database. To illustrate the point, we give the following example:

Figure 16. Incompatibility example 1.

Person
Name
Height
Age

Person Table:

Entity Name	Required?	Type
Name	Yes	Text 100
Height	Yes	Text 10
Primary key: Name		

Since there is no age attribute in the person table, there is clearly no way of saving or loading this attribute in the person class. One way of solving this problem would be to add an extra table having those attributes missing and the primary key of the table in where they are missing. In this way the missing attributes can be both retrieved and saved by joining the two tables.

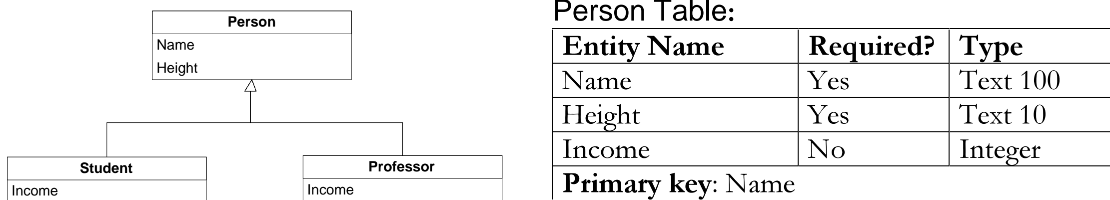
This solution only applies where there are no other systems that create new records in the old person table that our new application uses. If this is the case, and we are not allowed/able to change these systems, then clearly records added by these systems will not create a record in our added table, and the problem returns. So depending on the

situation we can either solve the problem by adding a table, or we can remove the offending attribute from the object model.

Another likely problem is generalisation hierarchies. In section 0 we saw how we could represent the type of the subclass by an attribute explicitly denoting where in the hierarchy the class was located. We also discussed that we often alternatively could test whether attributes found in only one of the subclasses where null, and from this deduce which subclass the record represented.

In some table structures we will not have the information required to find the type of the subclass. An example is the following small hierarchy and a corresponding existing database table where the data is supposed to be found.

Figure 17. Incompatibility example 2.



Person Table:

Entity Name	Required?	Type
Name	Yes	Text 100
Height	Yes	Text 10
Income	No	Integer
Primary key: Name		

As both students and professors have the same attributes – and only differ on their behaviour – we will not be able to determine whether a given record represents a student or a professor. Clearly the people having designed the database made no distinction between these two, and we will as a result of this not be able to do either.

Again we have to possible solutions. If the database can be changed we can add this attribute, and if not we will have to make only a person class modelling both students and professors.

6.2 INEFFICIENCIES

We above only dealt with changes that were required in the model due to incompatibilities in the two models, there can also be other reasons for changing the object model when interfacing to an exiting database. One major reason is efficiency.

Often relational databases will have a database scheme that not entirely reflects the data in the database, if the scheme has been optimised for faster operation on the data. This might lead to a situation where fetching data from the database for a given class in the model will involve joining a very large number of tables with a low efficiency as the result.

If this is very critical we must consider changing the object model. We should still be careful though, as from a strictly object oriented modelling point of view, this is certainly not good style. A good object model should reflect the problem domain of the system, and under normal circumstances not be changed for optimisation purposes.

6.3 SUMMARY

In summary we must admit that our current understanding of the potential problems with integrating an object model to an existing database is to little, as we only have a few guesses on how the existing database can make the transformation impossible or ineffective.

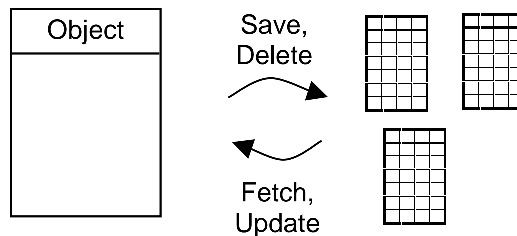
It seems likely that problems could occur when the connection has to be made, but it would be of relevance to investigate this aspect further to find more concrete examples.

7. HANDLING OUR DATABASE OPERATIONS

Having now discussed how to represent objects and relations between these in relational tables, and having discussed how using existing databases influences this, we can return once again to our four basic database operations, which we show again for easy reference.

The general idea is that we supply our storage mechanism with some kind of information telling it what the relation is between the classes in the object model, and the tables in the relational database. From this information, which we will call the data mapping, it should be possible to perform the operations shown below. In looking at the potential problems when executing the operations we should hopefully get a clear idea about what needs to be in this mapping.

Figure 18. The basic database operations



Save: The save operation takes as input a single object. If the object is already in the database, the data in the database corresponding to the object will be changed according to the values in the object. If the object is not in the database it is simply added to the database.

Delete: The delete operation takes as input a single object that has previously been fetched from the database, and deletes the data in the database corresponding to the object.

Fetch: The fetch operation takes as input a class and returns a list of all those instances of that class that are currently found in the database. To restrict the number of returned objects search criteria can be added.

Update: The update operation takes as input a single object that has previously been fetched from the database. It then compares the values in the object with the values in the database, and if the values in the database have been changed they will be fetched and the values in the object changed.

7.1 HANDLING FETCHES

7.1.1 Where to fetch what

When fetching instances of a class from the relational database our storage mechanism first of all needs to know where it can find the data for each and every attribute. Given the involved tables and attributes these can be joined together, the attributes can be read and copied to new objects. The first requirement for our data mapping therefore becomes that:

> *For every persistent class C, and for all attributes A of C, we must give a mapping showing in what table and what attribute A can be found. Furthermore it must be specified how the involved tables should be joined together.*

For faster fetching we can define a view per class involving those attributes that are used. From this view the attributes can then be read.

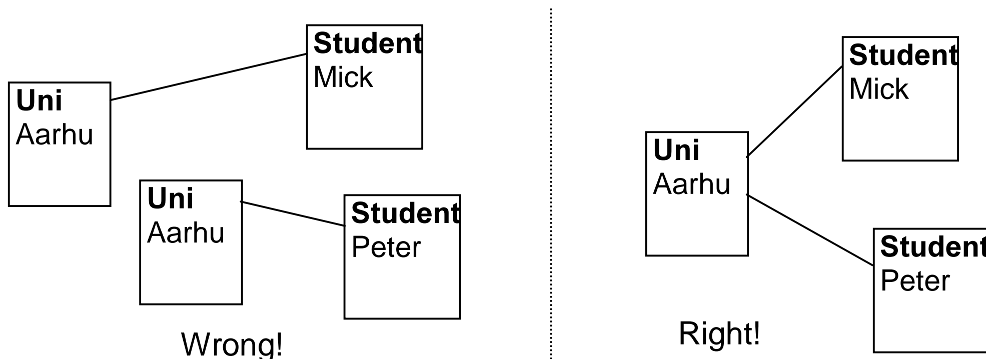
Finally in the special case where a single class has been spilt horizontically into several tables, we must provide enough information to decide in which of these tables the object is located.

7.1.2 Uniqueing

Another problem to be handled is that several fetches of the same object should only result in one instance in memory, which we will call *uniqueing* inspired by Enterprise Objects Framework (EOF) from NeXT [GMW94].

Take as an example the situation illustrated in Figure 19 below. On the left hand two instances of the same university object have been fetched, but since no uniqueing has been employed two instances has also been created in memory. The student object representing Mick currently refers to one and the student object representing Peter refers to one. This is of course an unwanted situation, as should for example one of the university objects be changed by accessing it from a student object, these changes will not be reflected in the other. The right side of the figure shows the situation with uniqueing, and as seen there is only one instance of the university object.

Figure 19. The uniqueing problem



We therefore need to consider how we can implement uniqueing in our implementation. Since every persistent object consists of one or more records, and since each and every record can be uniquely distinguished by the primary key, we can use these primary key's as a unique id on our objects. We can then test if an object with the same unique id as the one about to be fetched has been fetched before, and if this is the case use the one already fetched.

There is a problem though as our objects will not always contain all the attributes that forms the primary keys of all the involved tables for the object. Or even if this is the case, then the attributes of the object could easily have changed since the object was first fetched. We therefore need to save these primary key attributes somewhere outside the object. We suggest that for each class that is persistent a list is maintained, which we could call the *objectId* list. This list should contain a number of pairs, a pair for each instance of the class where the first element holds the involved attributes that form the uniquely identifying primary keys for the instance, and the second element holds a pointer to the instance's location in memory.

This solves both our problems. When an object is about to be fetched we can search the list for a set of matching attributes. If such a set is found we must be referring to the same object, since these attributes form the primary keys for the class, and we simply return the previously fetched object by following the pointer in the pair. And since the values are kept in the list, and this list is not affected by changes to the attributes in the object we have also solved the second problem. If furthermore we somehow hash the values in the list, this look up should be quite fast.

Now we only need to know which attributes in the database have the property of uniquely identifying the instances of the class, so this will become the second requirement for our data mapping:

> *For every persistent class C, we must identify the set of database attributes that uniquely identify any instance of C.*

It can be observed that the combination of the primary keys in those tables that the class retrieves data from, will fulfil the above requirement, although there might be a smaller set with the same property.

A final problem related to uniqueing is what to do if we fetch an object, which has previously been fetched. To support uniqueing we said that we should just return the reference to the object already fetched. But what if the state of the object has changed in database? We can do a number of things. We can read the new state in the database, and overwrite the contents in memory. This might not be so good though, as any changes to the object in memory will hereby be lost. Alternatively we could ignore that the contents have changed on disk, but this might not be optimal either if the object requesting the new object needs the new state. We could also do a compromise. When an already fetched object is requested to be fetched again, we test whether it has changed on the disk. If this is the case we throw an event on the object in memory saying that it has changed on disk, and only change the object after the event has returned. In this way at least a warning is given before the object changes.

7.1.3 How to handle relationships

We also need to consider how we will handle relationships. Say that we fetch a university object that is associated to 5000 student objects. Should we then fetch them all, as they can be accessed from the university object? This would be very expensive, and as these could easily be further related to other objects, we could in the most extreme case end up fetching the whole content of the database. This is clearly not what we want.

Object oriented persistence solutions often instead use so called *lazy fetch*. The essence of this is that we do not fetch related objects, until they are actually used. In the example above we postpone the fetching of a student object until the association from the university object to it is followed.

We then only need to consider how we will resolve the relationship when the relation is followed. One strategy is the one employed by the before mentioned EOF system from NeXT. In this system for every related object a so-called fault object is created. This object acts as a stand-in for the real object, in the sense that it does not hold the data of the real object, but instead hold those attributes that are needed to fetch the real object. These attributes are of course the before mentioned uniquely identifying attributes, that we also used to achieve uniqueing. This strategy saves the time it would take to fetch all the related objects, but still consumes the same amount of memory, as the fault objects in EOF use the same amount of memory as the real objects they act as stand-in's for.

Another alternative – which is cheaper in memory consumption – would be to only store those values that are necessary to fetch the related objects. As these values are exactly those we saved for fetched objects in our objectId list, we could appropriately store them in this list for this purpose as well. When a reference to a related and not yet fetched object is followed, we have to intercept this, fetch the instance using the information in the list and redirect the call to the fetched object.

7.2 HANDLING UPDATE

Above under the fetch operation we considered problems that could occur when fetching an object that had previously fetched, in our discussion of uniqueing. These considerations of course apply equally well to the update operation, as this operation does just that: Fetches an object again that has been fetched earlier.

As there seems to be no difference between these two situations, we will not discuss this operation any further.

7.3 HANDLING SAVE

When saving objects, we can experience further complications not seen when fetching objects. First of all we need to determine whether the save operation is requested on an objected that has been previously saved, or on a new object. This can be done by scanning our objectId list for an entry referring to the object to be saved. If such an entry is found, the request is to save the state to the existing object in the database. We can then locate the existing object using the uniquely identifying attributes in the pair in the list, and save the changed attributes to the database.

If the object is not found in the list, is has not been fetched during this execution so we assume that it does not exist in the database and create a new object in the database. When posting this new object there is of course the possibility that an equivalent object was already in the database from an earlier execution, but this should return an error from the database which we then just have to pay attention to.

Another problem is if the object contains attributes, which there are no attributes for in the tables. We already treated this problem in section 6.1 on Incompatibilities. We here concluded that either the attributes must be removed from the object model, or additional tables containing the extra attributes must be added.

Finally it can present a problem that the tables that we save the object to, can contain required attributes for which there are no attributes in the object. This is a serious problem, as this will make it impossible to post the records to the database. We have two choices in solving this: Remove the attributes from the table, or somehow enter a value in the attributes. As our philosophy was not to change the tables, we will recommend the last solution. The best way to do this would be to give default values in our data mapping for those required attributes for which there are no data in the objects. This will be our third requirement for the data mapping:

> *For every persistent class C, that involves the set of tables T in the database, we must for the required attributes A in T either give a mapping from attributes in C or default values.*

7.4 HANDLING DELETE

The delete operation presents no serious problems. As we already have the uniquely identifying attributes in the objectId list, we can easily construct an SQL-delete expression using these key values.

There are a number of smaller considerations though. First of all we must deside whether we should dispose of the memory instance on which the delete operation is invoked. There seems to be no reason for this, as we might wish to keep operating on the object in memory, even though it no longer exists in the database, so we will leave this up to the programmer.

Furthermore we need to consider what to do when deleting objects that are related to other objects via relations such as association and aggregation. In [AJK93] they describe three kinds of deleting: *Block*, *propagate* and *remove*.

In a delete with block, it is first examined if the object is related in some way to other objects. If so the deletion is considered potentially harmfull, and the delete operation is cancelled or “blocked”.

When deleting with propagate all those objects that are related to the object which is to be deleted are also deleted. In this way the full transitive closure of objects that can be referenced from a single object can be deleted quickly.

Finally when deleting with remove, only the object in question is deleted, but all references in the database – such as foreign keys – to the object are set to null.

7.5 SEARCHING FOR OBJECTS

Our definition of the fetch operation vaguely said that we should be able to restrict the persistent instances of a given class to retrieve during the fetching, so we need to consider just how this should be expressed.

In both of our earlier mentioned solutions, *EOF* from NeXT [GMW94] and the *RDBobj* solution described in the thesis of Broløs and Pilgaard [BrPi95] they use some form of SQL statements.

In EOF the so-called *qualifiers* can be added to instances of the class that represents a database table. These consist of a format string with the type (<, > or =) and a number of arguments, and using these the `WHERE` clause of the select statement can be formed by the EOF system. An example is a qualifier with type "%x < %y", and arguments "Salary" and "25000", would restrict the fetch to the records having a values higher than 25000 in the "Salary" attribute.

In RDBobj, each class is represented by a *RDBobjview* class, which has a scan method, where a *SQLcond* variable can be set. This scan method will return all objects corresponding to a `SELECT` statement with the contents of SQLcond placed after `WHERE`. Furthermore the instances that are created when fetched are compared to an *OBJcond* condition, and only those that fulfil this condition will be returned as the result.

While both these solutions indeed do the trick, still we do not consider them optimal. The main problem is that the application programmer cannot search for data without knowing what is considered legal in the SQL `WHERE` clause and knowing what the database looks like.

The problem with not knowing SQL is not too bad – after all there is not much to know. But having the restriction that the application programmer should also know the fine details of the database tables is a heavy burden. Optimally the programmer should only think in terms of the object model, and not worry about how this model is mapped to the database.

This does not seem to hard achieving either. As our first requirement for the data mapping was that for every attribute in the classes it should say where these could be found in the database, it should be possible to have the programmer express his conditions in terms of the class attributes and then change this into SQL using the information in the mapping. This would be a large step forward, and would furthermore have the advantages that a change of the database would only require a change of the data mapping, and not all program statements involving database searches.

Another alternative could be Object SQL as suggested by the ODMG (Object Database Management Group). Object SQL is a superset of SQL92, that allows searches to be expressed in a way so that they resemble OO language primitives, while at the same time is (almost) totally compatible with SQL [CBB+97].

7.6 THE DATA MAPPING

We have now several times mentioned the data mapping, and the requirements to it, so it seems that we should formalise further just what we are talking about.

Both of our mentioned systems EOF [GMW94], and RDBobj [BrPi95] use a sort of data mapping. In the case of EOF the mapping is quite simple, as the OO-model in EOF is constructed directly from the database definition, and there is always a one class to one table relationship. Only minor adjustments such as leaving out attributes, changing the names and adding simple attributes that are found in other classes are possible.

The workflow in EOF is that the database is defined first. After this the EOModeller tool is started, which reads the database definition, allows the above described minor adjustments and then the tool generates a template object model.

In RDBobj, the approach is to start with both the database and the object model, and the make the mapping from these. They use a textual mapping defined by a BNF, such as:

```
classdef person
       cpr:   PERSON.cprNo,
       name:  PERSON.name,
       age:   PERSON.age;
```

which basically describes that instances of the person class, can find their `cpr` attribute in the attribute `cprNo` of the `PERSON` table, and so on.

We suggest an approach close to the one of RDBobj. We should start by defining the database and the OO-model, and then make a mapping describing the relation between these. This mapping should conform to the requirements that we found in the last section, which we show again below, for easy reference.

Figure 20. Data mapping requirements

For every persistent class C, and for all attributes A of C, we must give a mapping showing in what table and what attribute A can be found. Furthermore it must be specified how the involved tables should be joined together.

For every persistent class C, we must identify the set of database attributes that uniquely identify any instance of C.

For every persistent class C, that involves the set of tables T in the database, we must for the required attributes A in T either give a mapping from attributes in C or default values.

This mapping should probably be defined as a textual format by a BNF as in the case of RDBobj, but furthermore one could integrate the mapping process into other tools used in OO development. For example it might be promising to have a tool that read the database definition and presented the tables in a graphical format along side the OO-model, and then one could construct the mapping in a more or less graphical manner by dragging connectors from one to the other.

8. SUMMARY

We started out with a number of general considerations about persistent storage, after which we continued on to describing the problems that might arise when mapping OO-models to relational databases.

We showed a number of mappings that could be used when mapping OO-models to relational databases and discussed advantages and disadvantages of each of these. We then discussed how these were influenced when the mapping was to be done to an existing database, where we found problems such as attributes missing in the tables, generalisation hierarchy problems and inefficient fetching.

After this we introduced four general database operations and described the problems that would arise when implementing these, such as uniqueing, maintaining an id-list, handling relationships and how to express searches.

Finally we looked at the data mapping, and made a "loose" suggestion for its syntax and how it should be generated.

9. BIBLIOGRAPHY

[ARK93] Shailesh Agarwal, Richard Jensen, Athur M. Keller. *Bridging Object-Oriented Programming and Relational Databases.* Appeared in SIGMOD, May 1993. [1]

[AKK95] Shailesh Agarwal, Christopher Keene, Arthur M Keller. *Architecting Object Applications for High Performance with Relational Databases.* Appeared in "OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance, Austin, TX", October 1995.

[BEL+91] Michael Blaha, Frederick Eddy, William Lorensen, William Premerlani, James Rumbaugh, *Object-Oriented modeling and design*. Prentice-Hall, New Jersey, 1991.

[BrPi95] Jette Holm Broløs, Kristian Bang Pilgaard. *Objekter i relationsdatabaser.* Masters Thesis (in Danish) from Institute of Computer Science, University of Copenhagen (DIKU). 1995.

[CBB+97] Edited by R.G.G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. *The Object Database Standard. ODMG 2.0.* Morgan Kaufman Publishers. 1997.

[GMW94] Nik Gervae, Katie McCormick, Greg Wilson. *Enterprise Objects Framework Developer's Guide*. Nextstep Developer's Library, release 3. NeXT Computer, Inc., 1994.

[KoSi91] Henry F. Korth, Abraham Silberschatz. *Database System Concepts.* McGraw-Hill, 1991.

[Mjø94] *The Mjølner BETA System – Persistence in BETA.* Mjølner Informatics Report, Aarhus, 1994.

[MMN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison Wesley, Wokingham 1993.

[1] The Arthur M. Keller articles can be found in postscript format at:
`http://www-db.stanford.edu/pub/keller/#Object`