# Mapping Objects to Tables

## A Pattern Language

Wolfgang Keller
c/o EA Generali, Neusetzgasse 1, A 1100 Wien, Austria
Email: 100655.566@compuserve.com
http://ourworld.compuserve.com/homepages/WofgangWKeller/ ; http://www.sdm.de/g/arcus/

## Abstract

Mapping Objects to Tables is a problem that has occurred as long as people want to program in an object oriented language but have to use relational instead of object oriented databases for some reasons. Mapping Objects to Tables is only one family of problems that occurs in object/relational access layers. The whole context of object/relational access layers has been described by many authors (see [Bro+96, Col+96]) and is subject of our own future work.

## Introduction

Object-orientation and the relational model are different paradigms of programming. When objects need to be stored in relational databases, the gap between the two different sights needs to be bridged. If only data abstraction modules have to be mapped to a relational database, life is comparably easy [Kel+97]. With full blown object models the concepts of object oriented programming have to be mapped to relational table structures. These are:

- Aggregation,
- inheritance and polymorphism,
- associations between classes,
- and data types – smarter than SQL data types.

Each of the above concepts may be mapped using different solutions for the same problem. We will describe each different solution as a separate pattern. This approach allows us to clearly demonstrate the consequences of using a solution with respect to the general forces presented next. Mapping data types requires a larger effort [Kär95] and may be treated as a pattern language of its own. We will therefore leave it to future pattern mining work.

## General Forces

- *Performance:* One of the major forces that you should take into account when mapping objects to tables is performance. The way objects are mapped to tables has significant influence on the number of database accesses that occur in a system. Database accesses that have to be executed using hard disks or other external media are measured in milliseconds ($10^{-3}$ sec.). Processor cycles on the other hand are measured in nanoseconds ($10^{-9}$ sec). It is therefore a good idea to waste a few processor cycles and some RAM memory to economize on slow IO.

- *Read versus write/update performance*: The solutions we will present for various mapping problems have different characteristics if it comes to read versus update/write performance. Some mappings allow you to read everything needed in a single database access while it takes several database operations to write the same object, due to the mapping used for inheritance. Therefore be sure you know the frequency of read and write/update operations before you commit yourself to a certain table design.

- *Flexibility and maintenance cost*: Sometimes you might want to use a database mapping in a prototyping process. In this case flexibility is more important than performance as you will often insert or delete attributes, add or delete classes and restructure your class hierarchy. Once the hierarchy and classes become stable you may want to switch to a mapping with optimal performance.

- *Performance and redundancy versus maintenance cost and normal forms*: The relational calculus helps you eliminate redundancy using normal forms and factorization. Relational database applications on the other hand show best performance with a minimal number of accesses to the database. The expensive factor is seek time for a certain record and not bandwidth for reading the data from a disk once they are located on the disk. Hence these applications perform best if they are able to retrieve all data needed for a use-case with a single access to the database or if they hit a cluster of data.

    - Accesses to the database can be reduced by eliminating factorization and ignoring normal forms – which has negative consequences on the maintainability of the application [Kel+97].

    - Clustering can be influenced by database administration.

  Maintainability of a data model and performance are two conflicting goals. Therefore: The harder you optimize your data model for performance, the higher your maintenance cost in case of changes to the application. Redundancy and all kinds of anomalies, normally prevented by the use of normal forms, have to be taken care of by maintenance.

- *Space consumption versus Performance*: There are mappings that use no surplus database space (fields with null values and the like) and others that leave large portions of a database record unused. It is not surprising that the worst space hogs are often the fastest performers.

- *Query processing*: There are two conflicting purposes, data have to serve in a business information system.

    - Data have to be ready for online transaction processing with good performance. This implies restructuring of data for optimal performance. See [Kel+97] on how to optimize table schemes for good performance.

    - Data have to be ready for data warehouse purposes. This implies that data have to be represented in a form that is well suited for ad hoc queries. Normal forms, no redundancy and maximum factorization serve this purpose.

  Building a data warehouse often implies separating the queryable data from the data needed for fast online processing. If you design a table mapping for objects, check whether the application is a data warehouse or an online processing application.

- *Application style*: Besides database driven business information systems there are other types of information systems. Using a relational database as persistence mechanism for some of these might end in disaster. Some examples are.

  - *CAD applications*: CAD applications are used to manipulate large sets of very complex, interrelated objects. Transactions are long. A CAD designer typically checks out a design, works on it for hours and then checks it back into some data store. Building such applications on top of a relational database, using an object relational database mapping is doomed to fail. Simple pointer dereferencing in working storage is faster by a factor $10^6$ than joins. Relational databases are not intended for very long transactions with zero collision rate.

  - *CASE Tools*: CASE tools have characteristics similar to CAD systems. IBM's negative experience with the AD/Cycle repository is a prominent example of what happens if such applications are implemented on top of a relational database.

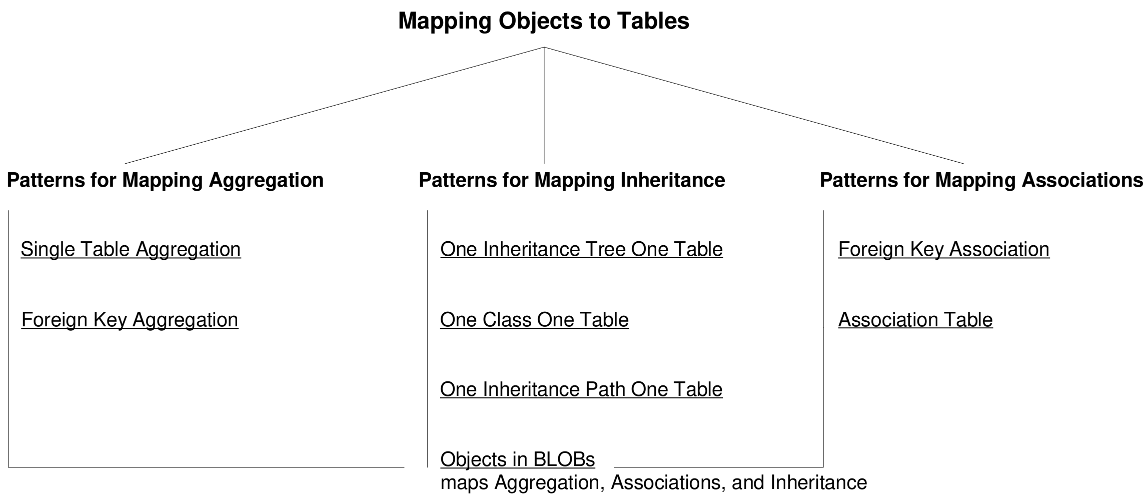
  - *Any check in / check-out persistence applications*: The above example can be generalized to applications that use complex, interrelated objects, allow direct manipulation and allow the user to check them out of a database for a longer period of time. Such systems should be built using non-relational data stores.

  Check you do not build one of the above applications before you map objects to relations.

- *Integration of legacy systems*: Business information systems are seldom developed from scratch. Instead, you have to connect to legacy systems, which you are not allowed to touch. In our case you might have to build objects on top of legacy data. In this case you have to use whatever mapping patterns that fit your legacy data. You may then apply the consequences sections of the patterns. They inform you about performance implications of the mappings you have to use.

## Roadmap of the Pattern Language

This pattern language is structured according to the problem structure. There are alternative solutions (patterns) for each of the three problems: Mapping aggregation, inheritance and associations.




## Cookbook Aspect of the Pattern Language

This pattern language presents alternative solutions to the three main problems you encounter, when mapping objects to tables. Depending on your project's requirements you might want to optimize your mapping for flexibility, easy maintenance, low database space consumption, or performance. To give you a first impression, we have listed the patterns together with their significant consequences in a matrix. The matrix will help you with a first guess where to start reading. But the matrix cannot provide a detailed technical discussion like the patterns.

| Pattern | Performance | | | Space Consumption | Flexibility, Maintainability | Ad-hoc Queries |
|---|---|---|---|---|---|---|
| | Write/ Update | Single Read | Polymorphic Queries | | | |
| Single Table Aggregation | + | + | * | + | - | - |
| Foreign Key Aggregation | - | - | * | + | + | + |
| One Inheritance Tree One Table | +o | +o | + | - | + | + |
| One Class One Table | - | - | -o | + | + | - |
| One Inheritance Path One Table | + | + | - | + | - | - |
| Objects in BLOBs | +o | +o | o | + | - | - |
| Foreign Key Association | - | o | * | + | + | + |
| Association Table | - | o | * | + | + | + |
| + good, - poor, * irrelevant, o depends, see detailed discussion | | | | | | |

## Patterns for Mapping Aggregation

The question, when to use aggregation and when to use associations is very hard to answer at the object oriented modeling level (OOA). For our purposes the answer to this question is mostly influenced by performance or flexibility considerations. You may use Single Table Aggregation, which is the natural way to map aggregation. You may also use Foreign Key Aggregation which is the usual way to map 1:n associations and is also discussed as Foreign Key Association in the section on mapping associations.

### Pattern: Single Table Aggregation

#### Abstract

The pattern shows how to map aggregation to a relational data model by integrating all aggregated objects' attributes into a single table.

### Example

Consider the following object model:

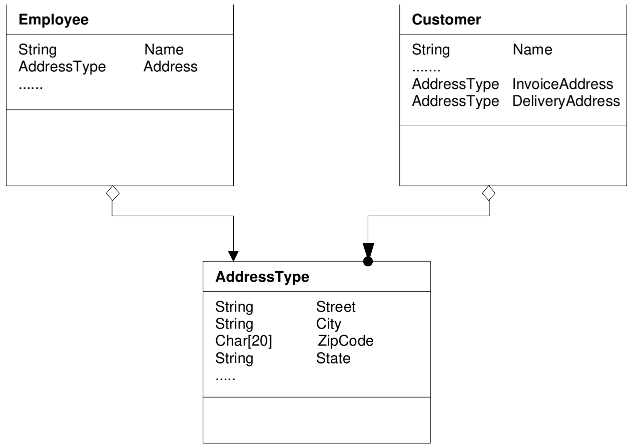



Figure 1: An AddressType aggregated by more than one other types

### Problem

How do you map aggregation to relational tables?

### Forces

- *Performance:* For optimal performance the solution should allow to retrieve an object with one database access without any join operations. Database accesses should fetch a minimum number of pages to economize on I/O bandwidth.

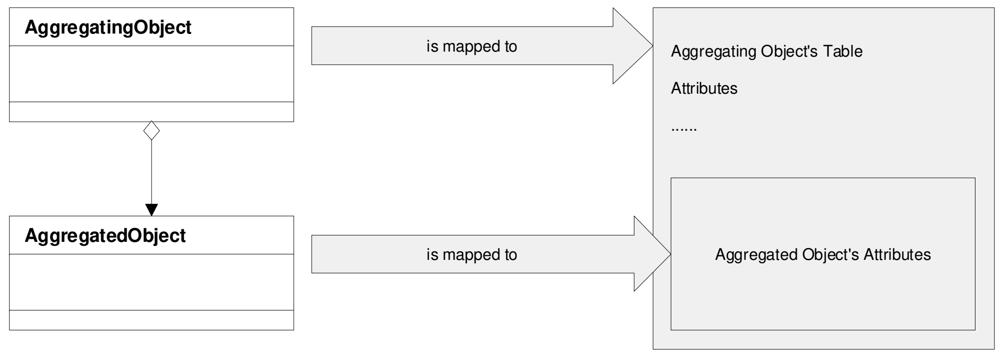
- *Maintainability:* For optimal maintainability, aggregated types, that are aggregated in more than one object type, should be mapped to one set of tables instead of being strayed identically across many different spots in the physical data model. Normalization should be used at the data model level to ease maintenance and ad hoc queries.

- *Consistency of the database:* Aggregation implies that the aggregated object's life cycle is coupled with the aggregating object's life cycle. This has to guaranteed by either the database or application code

### Solution

Put the aggregated object's attributes into the same table as the aggregating object's.

### Structure




The AggregatingObject is transformed to a table of the physical data model. The AggregatedObject's Attributes are integrated into that table.

### Example Resolved

To resolve our above example we look at the table created for the Customer object. The InvoiceAddress and the DeliveryAddress are both integrated into the Customer's database table.


Figure 2: Mapping an aggregated object type into the aggregating object's table

A prefix is used to distinguish the attributes. This is similar to the resolution of structures in C++, using a dot notation (like Customer.DeliveryAddress.Street).

### Consequences

- *Performance:* The solution is optimal in terms of performance as only one table needs to be accessed to retrieve an aggregating object with all its aggregated objects. On the other hand, the fields for aggregated objects' attributes are likely to increase the number of pages retrieved with each database access, resulting in a possible waste of I/O bandwidth.

- *Maintenance and flexibility:* If the aggregated object type is aggregated in more than one object type, the design results in poor maintainability as each change of the aggregated type causes an adaptation all of the aggregating object types' database tables.

- *Consistency of the database:* Aggregated objects are automatically deleted on deletion of the aggregating objects. No application kernel code or database triggers are needed.

- *Ad-hoc queries*: If you want to form a query that scans all AddressType objects in the database, this is very hard to formulate.

#### Implementation

- *Naming convention:* You need to think of a prefix or another naming convention for the aggregated object's attributes that appear in the aggregating object's table. In the above example we use a prefix that is a short form of the attribute name.

- *Physical database page size:* The positive effects of aggregating an object in the same table can be partially compensated if the aggregated object's attributes start a small fraction of a new database page. In such a situation two database pages need to be read instead of one.

#### Variants

We have discussed the simple case of a 1:1 relation between aggregating object type and aggregated object type. See Foreign Key Association for how to map a 1:n relation between aggregating object and aggregated object. See also Overflow Table [Kel+97] for a trick to avoid using foreign key associations in case of 1:n relations.

#### Related Patterns

Foreign Key Aggregation is an alternative solution to Single Table Aggregation. See also Representing Collections in a Relational Database [Bro+96]. When applied to ordinary relational database access layers, it can be compared to Denormalization [Kel+97].

#### Further Reading

"*Mainstream Objects*", a book by *Ed Yourdon et al.* [You+95] dedicates its whole chapter 21 to the question of when and how to use aggregation versus associations at the modeling level.

## Pattern: Foreign Key Aggregation

#### Abstract

The pattern shows how to map aggregation to a relational data model using foreign keys.

#### Context

Reconsider the example for Single Table Aggregation (see Figure 1). Presume you want a solution that treats the AddressType as a first class object and that allows better maintenance than Single Table Aggregation.

#### Problem

How do you map aggregation to relational tables?

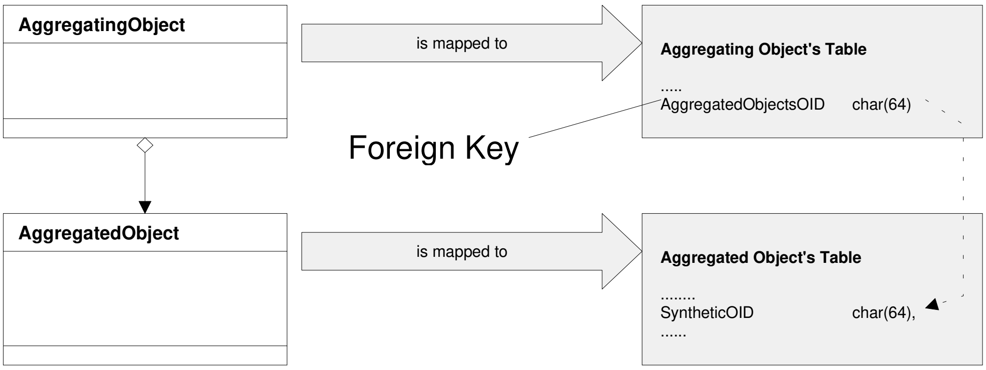
## Forces

See the Single Table Aggregation pattern.

## Solution

Use a separate table for the aggregated type. Insert an Synthetic Object Identity into the table and use this object identity in the table of the aggregating object to make a foreign key link to the aggregated object.

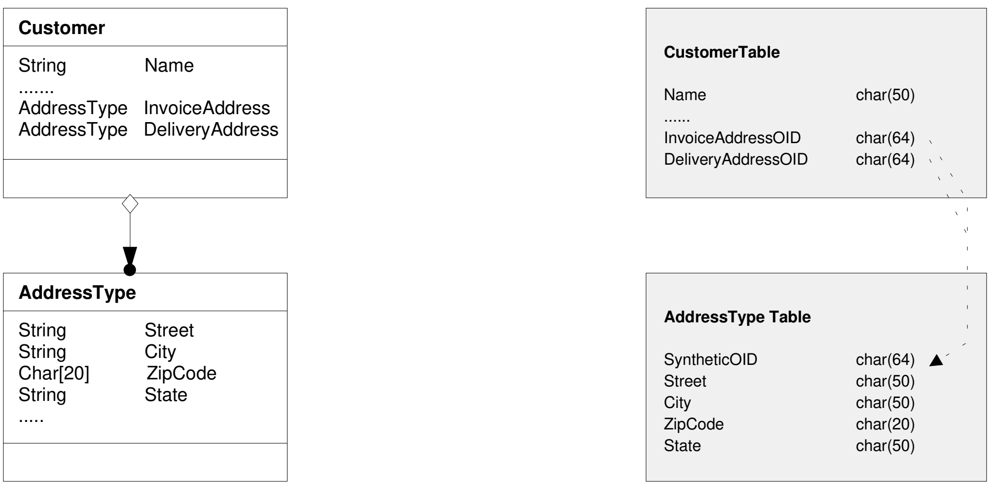
## Structure




The AggregatingObject is mapped to a table. The AggregatedObject is mapped to another table. The Aggregated Object's Table contains a Synthetic Object Identity. This SyntheticOID is referenced by the AggregatedObjectsOID foreign key field in the Aggregating Object's Table.

## Example Resolved

If we apply the solution to the example on page 5, we get a Customer Table that contains two foreign key references to the AddressType Table . The AddressType Table contains a Synthetic Object Identity field that is used to link the two tables.



Retrieving a customer object from the database now costs three database access operations (one for the Customer and one for each AddressType, Invoice Address and DeliveryAddress) instead of one in the case of Single Table Aggregation.

This can be brought down to a single join database access, if the AddressType Table is equipped with an additional back link field that points to a Synthetic Object Identity of the Customer Table. The cost of this is getting a result set of two addresses, each with all the customer attributes.

#### Consequences

- *Performance:* Foreign Key Aggregation needs a join operation or at least two database accesses where Single Table Aggregation needs a single database operation. If accessing aggregated objects is a statistical rare case this is acceptable. If the aggregated objects are always retrieved together with the aggregating object, you have to have a second look at performance here.

- *Maintenance:* Factoring out objects like the AddressTypes into tables of their own makes them easier to maintain and hence makes the mapping more flexible.

- *Consistency of the database:* Aggregated objects are not automatically deleted on deletion of the aggregating objects. To perform this task you have to provide and maintain application kernel code or database triggers. This is also an implementation issue. You have to chose one of these two options.

- *Ad-hoc queries:* Factoring out aggregated objects into separate tables allows easy querying these tables with ad-hoc queries.

#### Implementation

- Consider using domain keys instead of Synthetic Object Identities. Domain keys have the drawback that they cannot be used for arbitrary links pointing back to an owner object as the owned object type cannot know all types of objects that will ever own it.

- Consider inserting a link back from the aggregated object to the aggregating object. In our address example this is accomplished by inserting a field into the address table that stands for the owner of the AddressType object. As the owner may be an Employee, a Customer or some other type that aggregates the AddressType you have to use a Synthetic Object Identity as the link's type. Bi-directional links offer some advantages for queries, consistency checking and other purposes. You don't have to search the aggregating object's table to find an owner of an aggregated object. On the other hand, backlinks are more expensive in terms of database operations needed to keep them up to date.

#### Related Patterns

For an alternative see Single Table Aggregation. Foreign Key Association works very similar. See also Representing Collections in a Relational Database [Bro+96].

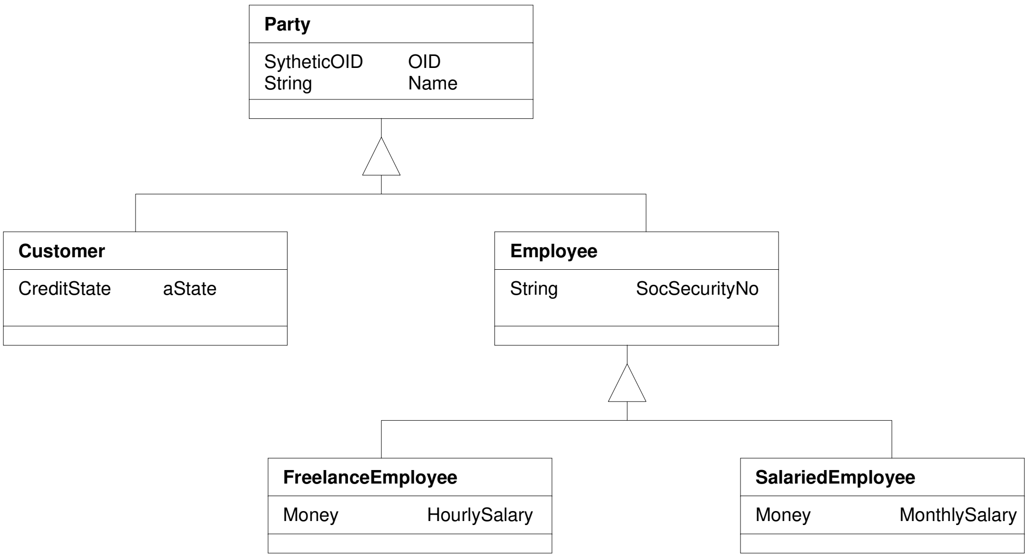
## Patterns for Mapping Inheritance

There are various ways to map inheritance hierarchies to relational database tables. The patterns presented below are pure forms of exactly one mapping style. In practice you may mix mapping styles to arbitrary table mappings.

The following discussion does not cover multiple inheritance. There are few meaningful examples of domain level multiple inheritance anyway. Most uses for multiple inheritance are motivated by protocol inheritance. A class inherits several protocols from abstract base classes. Protocol classes that have a persistent state are rare. So simple inheritance covers most practical cases.

## Running Example

As a running example we use a part of a so called partner system. A Party is any form of person (natural person or institution) our company has to work with. Customers are Parties as well as Employees. When it comes to Employees, we distinguish between SalariedEmployees and FreelanceEmployees. This results in the following object diagram.




We did not add all attributes needed for a real life application but insert only as many attributes as are needed to demonstrate the different mapping patterns that we discuss. Therefore we do not use any complex attributes or relationships. For our example we assume that none of the classes is an abstract base class. All five classes may have instances.

## Pattern: One Inheritance Tree One Table

### Abstract

The pattern demonstrates a way to map a complete inheritance hierarchy to a single database table.

### Problem

How do you map an inheritance hierarchy of classes to database tables?

### Forces

The forces relevant here, besides the General Forces on page 1, are:

- *Polymorphic read and space consumption versus write/update performance*: In an inheritance hierarchy you need to support queries for *all* Party objects matching some given criteria. The result set is polymorphic, In our example it might contain Employees, FreelanceEmployees or

SalariedEmployees. Solutions that best support polymorphic queries are those who either waste disk space or are expensive in terms of write performance.

- *Locking schemes of your database*: Some databases implement page level locking only or might be programmed to escalate locks very early. In this case you have to take care that database traffic on a single table does not exceed a limit that results in excess locks and poor performance. If you map to many classes to a single table, it is likely that you attract much traffic.

- *Depth of the inheritance hierarchy*: Some solutions that work acceptable with flat inheritance hierarchies become ugly with very deep inheritance hierarchies.

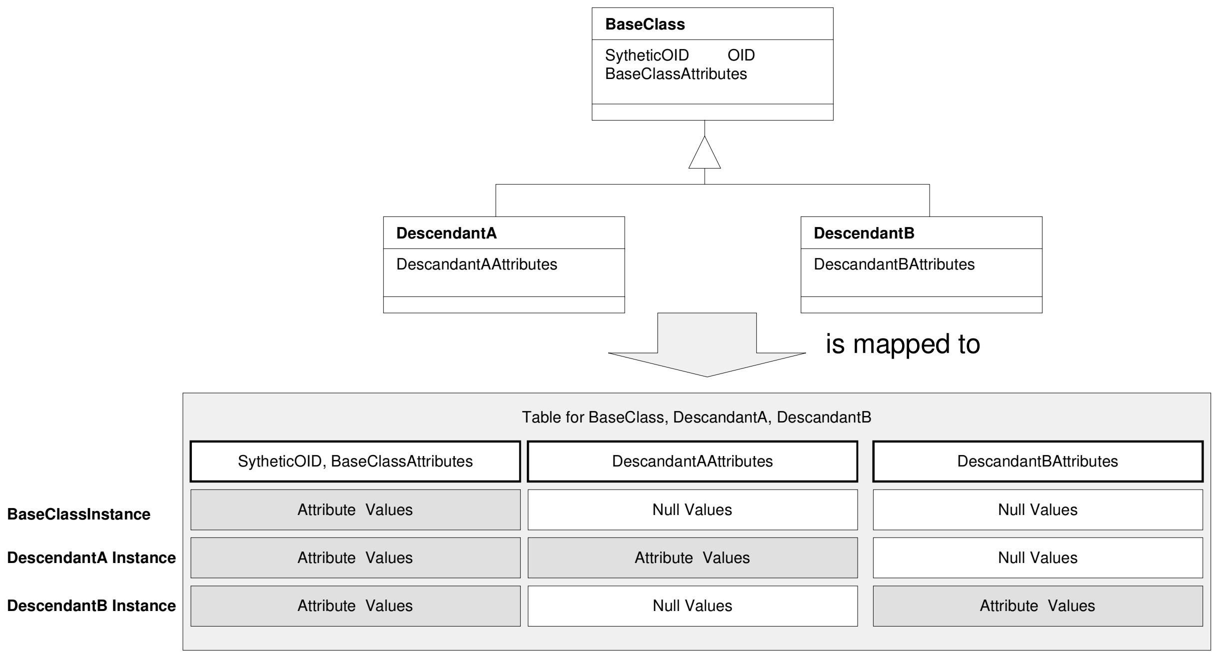
- *Maintenance effort*: Mapping solutions that clutter a single object's data across several tables might be fast for polymorphic reading. The drawback is, they are very hard to maintain in case new object attributes are added or existing object attributes are deleted. Schema evolution needs to take into account that data are replicated across the physical data model. This may easily turn into a maintenance nightmare. Other maintenance cases are insertion or deletion of a class in an inheritance hierarchy.

- *User-defined queries*: If you want to give your user the opportunity to form her own queries you need to assure that table mappings are still understandable from a user's perspective.

### Solution

Use the union of all attributes of all objects in the inheritance hierarchy as the columns of a single database table. Use Null values to fill the unused fields in each record.

### Structure



### Example Resolved

The table design for our running example looks as follows.

```
Table PartyHierarchy

// Party attributes
SyntheticOID          char(64)
Name                  char(50)
....
// Customer attributes
aState                char(5)
....
// Employee attributes
SocSecurityNo         char(15)
....
// FreelanceEmployee attributes
HourlySalary          numeric(5,2)
....
// SalariedEmployee attributes
MonthlySalary         numeric(7,2)
....
```

### Consequences

- *Write and update performance*: Using <u>One Inheritance Tree One Table</u> allows reading and writing of any BaseClass descendant with a single database operation.

- *Polymorphic read performance*: As all BaseClass descendants can be found in a single table, polymorphic reading is straightforward. The only challenge is to construct the correct object type for a selected database record. There are plenty of patterns for this task like <u>Abstract Interface</u> [Col96].

- *Space consumption*: As you see from in the mapping depicted above, storing the objects' attributes requires more space than absolutely necessary. The waste of space depends on the depth of the inheritance hierarchy. The deeper the hierarchy and the bigger the difference between the union of all attributes and the attributes of an average object, the bigger the waste of space.

- *Balancing database load across tables*: Mapping too many classes to a single table may cause poor performance. The sources of such problems can be found in database behavior:

  - If your database uses page level locking, too much traffic on a single table may severely slow down the access. Parts of the effect may be compensated by clever clustering. If traffic on a single table gets too heavy, expect performance degradation and also deadlocks.

  - Too many locks on a single table may result in lock escalation[1]. The number of locks that cause lock escalation is typically a parameter of relational database systems.

  - Some classes need secondary database indexes to speed up search. If you implement many classes in a single database table, you add up indexes on that table. Too many indexes on a single table cause updates to become very slow as all the indexes have to be updated.

---

[1] For a brief discussion of lock escalation see [Dat95, page 406].

- *Maintenance cost*: As the mapping is straightforward and easy, schema evolution is also comparably straightforward and easy as long as the inheritance hierarchy does not become too deep.

- *Ad-hoc queries*: As the mapping is intuitively clear, formulating ad-hoc queries is fairly easy.

### Implementation

- *Consider mapping all objects to a single table*: You may also use the mapping to store all object types in a single table - resulting in heavy traffic on the table. For small applications this is a feasible and very flexible approach.

- *Waste of space*: You might check, whether your relational database allows packing of NULL values. In this case the above mapping becomes more attractive as you avoid waste of space for NULL values.

- *Type identification*: You need to insert type information into your table. You could compute the type information from the NULL values. This is not too convenient as Synthetic Object Identities should contain type information anyway. Hence it is better to use a straight Synthetic Object Identity that contains type information.

### Related Patterns

See also Representing Inheritance in a Relational Database [Bro+96]

## Pattern: One Class One Table

### Abstract

The pattern discusses how to map each classes in an inheritance hierarchy to a separate database table.

### Problem

How do you map an inheritance hierarchy of classes to database tables?

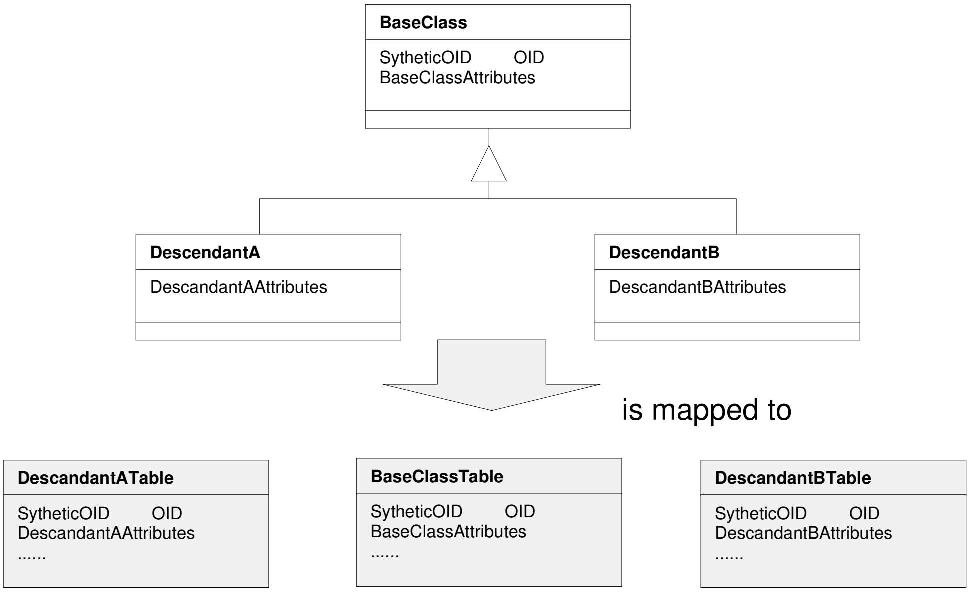
### Forces

The forces are identical to those discussed with the One Inheritance Tree One Table pattern.

### Solution

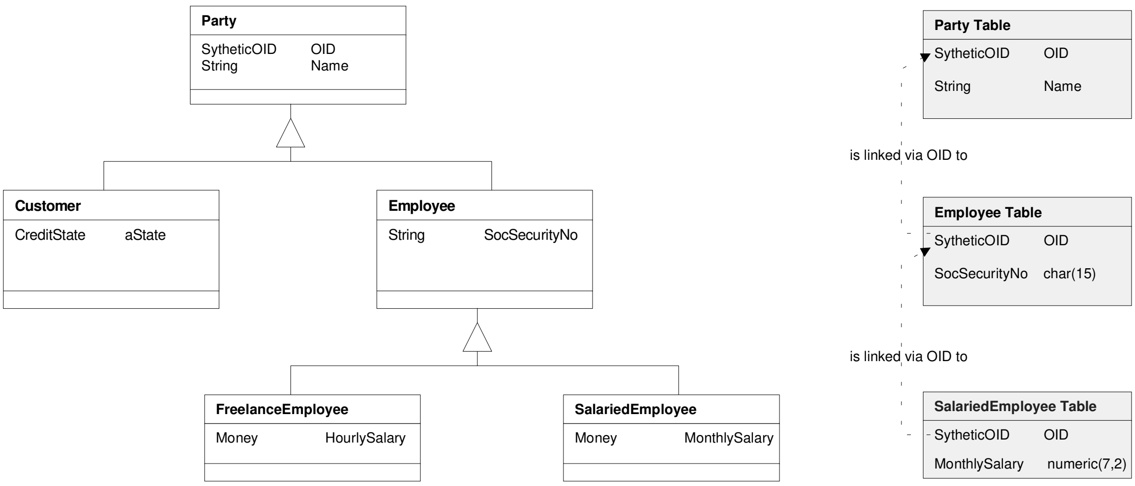
Map the attributes of each class to a separate table. Insert a Synthetic OID into each table to link derived classes rows with their parent table's corresponding rows.

**Structure**




**Example Resolved**

Mapping our running example to tables results in five tables - one for each class. A single instance of a SalariedEmployee is represented in three of these five tables.




**Consequences**

- *Write and update performance*: The pattern provides a very flexible mapping but is not the best performer. Consider reading a FreelanceEmployee instance in our running example. This operation costs 3 database read operations: One on the FreelanceEmployee table, one on the Employee table and also one on the Party table. Writing costs 3 database write operations, each updating one or more indexes. The mapping is expensive in terms of database operations for write- and update intensive tasks. The costs rise with the depth of the inheritance hierarchy.

- *Polymorphic read performance*: In our running example a FreelanceEmployee's instance has a corresponding Employee instance and also a Party instance in the respective tables. Therefore, polymorphic reading only require reading one table. This is one of the attractive sides of the pattern besides space consumption and

- *Space consumption*: The mapping has near optimal space consumption. The only redundant attributes are the additional synthetic OIDs needed to link the levels of hierarchy.

- *Maintenance cost*: As the mapping is straightforward and easy to understand, schema evolution is straightforward and easy.

- *Ad-hoc queries*: As the mapping generally requires accessing more than one table to retrieve an object instance's data, ad-hoc queries are far from straight forward but hard to formulate for inexperienced users.

- *Heavy database load on root tables*: The pattern causes heavy load on the root object type's table. In our running example, each transaction holding a write lock on the FreelanceEmployee table needs to be accompanied by a write lock on the Party and also on the Employee table. See the Consequences Section of One Inheritance Tree One Table for a discussion of the negative effects of tables that form a bottleneck.

### Implementation

- *Abstract classes*: Note that abstract classes are also mapped to a separate table.

- *Type identification*: For the above example we presume, that a Synthetic Object Identity contains type information. Some type information is needed to construct the accurate class from the result of a polymorphic read query.

### Related Patterns

See also Representing Inheritance in a Relational Database [Bro+96].

## Pattern: One Inheritance Path One Table

### Abstract

The pattern demonstrates a way to map all attributes occurring in an inheritance path to a single database table.

### Problem

How do you map an inheritance hierarchy of classes to database tables?

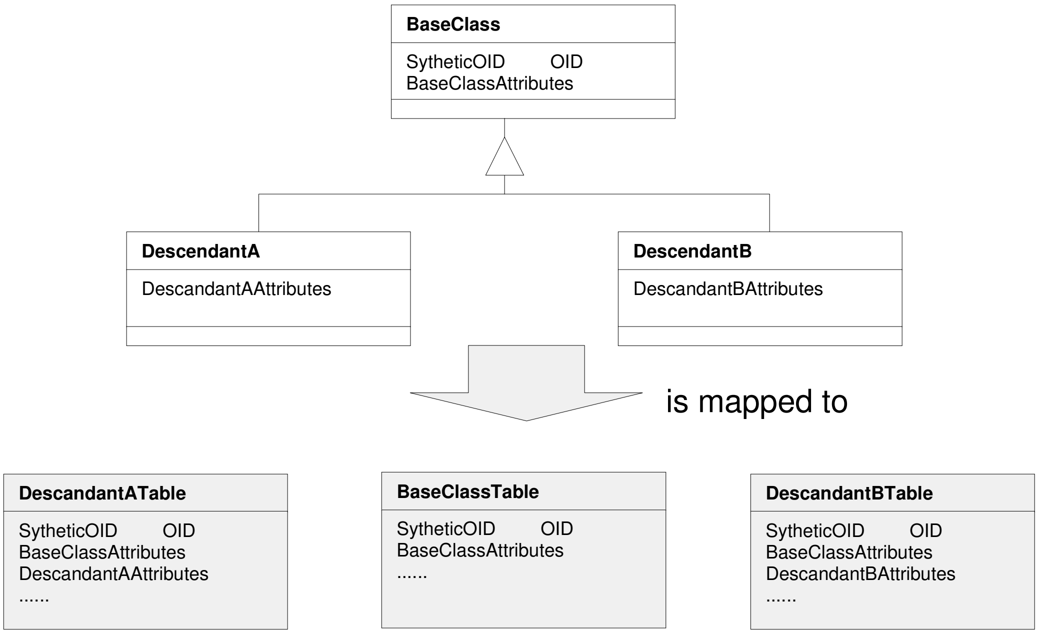
### Forces

The forces are identical to those discussed with the One Inheritance Tree One Table pattern.

## Solution

Map the attributes of each class to a separate table. To a classes' table add the attributes of all classes the class inherits from.

## Structure




## Example Resolved

Mapping our running example to tables results in five tables - one for each class. An instance of a SalariedEmployee is represented in one of these five tables. The SalariedEmployee is mapped as follows:

```
Table SalariedEmployee

// Party attributes
SyntheticOID                char(64)
Name                        char(50)
....

// Employee attributes
SocSecurityNo               char(15)
....

// SalariedEmployee attributes
MonthlySalary               numeric(7,2)
....
```

## Consequences

- *Write and update performance*: The mapping needs one database operation to read or write an object.

- *Polymorphic read performance*: A polymorphic scan of all Party objects in our running example would mean visiting 5 tables. This is expensive compared to One Class One Table or One Inheritance Tree One Table.

- *Space consumption*: The mapping offers optimal space consumption. There are no redundant attributes, not even additional synthetic OIDs in some ancestor's tables.

- *Maintenance cost*: Inserting a new subclass means updating all polymorphic search queries. The structure of the tables remains untouched. Adding or deleting attributes of a superclass results in changes to the tables of all derived classes. This may also touch polymorhic search queries if they are static rather than dynamically generated from a dictionary. Hence the pattern needs support by generators and dynamic queries to be maintainable.

- *Ad-hoc queries*: As the mapping generally requires accessing more than one table to perform polymorphic searches, ad-hoc queries for polymorphic search are hard to write for inexperienced users. Queries on leaf classes are trivial.

- *Database load on root tables*: There are no bottlenecks in tables near to the root of the inheritance hierarchy. Accessing an object exactly locks one table.

#### Implementation

- *Abstract classes*: Note that abstract classes are not mapped to tables.

- *Type identification*: You need not insert any type information into the tables as the type of an object cam be derived from the table name. Since Synthetic Object Identities should contain type information anyway, it would be a waste of effort to strip the type information to gain a few bytes of table space.

#### Related Patterns

See also Representing Inheritance in a Relational Database [Bro+96].

### Pattern: Objects in BLOBs

#### Abstract

The pattern demonstrates a way to map objects to a single database table using BLOBs. The pattern covers inheritance, aggregation, and associations. It is interesting from an academic point of view and as a source of ideas to solve unusual problems in mapping objects to relational databases.

#### Problem

How do you map objects to a relational database?

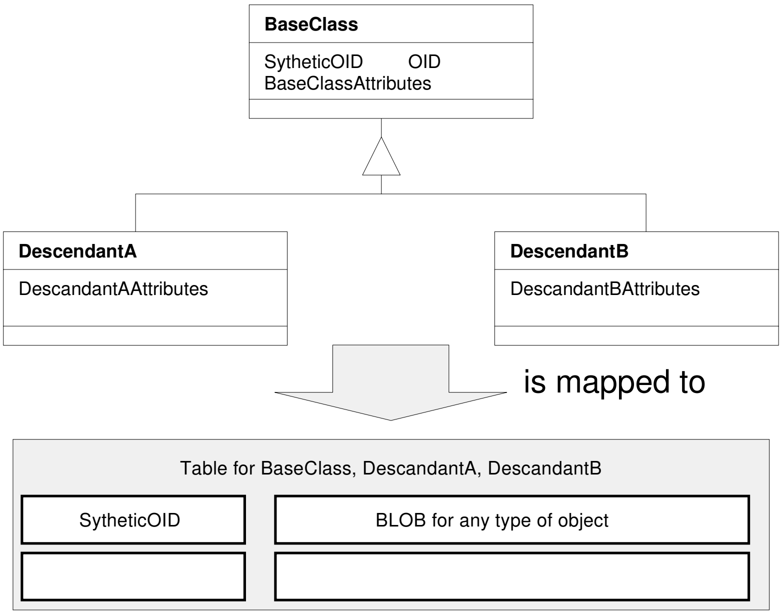
#### Forces

The forces are identical to those discussed with the One Inheritance Tree One Table pattern.

**Solution**

Use a table containing two fields: One for the synthetic OID and a second one for a variable length BLOB that contains all the data an object holds. Use streaming to unload the object's data to the BLOB.

**Structure**




**Example Resolved**

The table design for our running example or any other example looks exactly like the above table design.

**Consequences**

- *Write and update performance*: Objects in BLOBs allows reading and writing of any BaseClass descendant with a single database operation. Note that BLOBs are not the fastest way to access data types in many RDBMS.

- *Polymorphic reads*: Scanning classes for properties is difficult. As you do not have access to the internal structure of the BLOB you need to register functions with the database that give you access to the attributes. See [Loh+91] on how to implement such functions. Defining and maintaining these functions costs as much or even more effort as using database fields from the beginning.

- *Ad-hoc queries*: As scanning classes for properties is difficult, ad-hoc queries are also difficult to express. Again additional functions need to be defined.

- *Space consumption*: If your database allows variable length BLOBs, space consumption is optimal.

- *Maintenance cost*: Schema evolution is comparable to schema evolution in an object oriented database.

### Implementation

- *Sources of similar implementations*: Objects in BLOBs has been used in research prototypes. These tried to come as close as possible to a OODBMS using a relational database as storage manager. Hence implementing the pattern is very similar to implementing an OODBMS on top of a existing storage manager.

- *Balancing of database load across tables*: Mapping too many classes to a single table results in poor performance. For a discussion see the Consequences Section of the One Inheritance Tree One Table pattern.

- *Combining OODB properties with relational databases*: It is feasible to combine this pattern with other types of object/relational mappings. In this case the BLOB would hold complex object structures, like a project planning chart. Additional fields would hold the information needed to access the organizational data via normal ad-hoc queries (see Figure 3). Objects in BLOBs has been used in the SMRC research prototype [Rei+94, Rei+96]. The BLOBs in SMRC do not only hold a single classes' attributes but may contain a whole net of objects streamed to a BLOB (as depicted in Figure 3). The approach is used to allow coexistence of relational and OODBMS data in a single database. The approach is not exactly used in the pure form we describe in the above pattern.

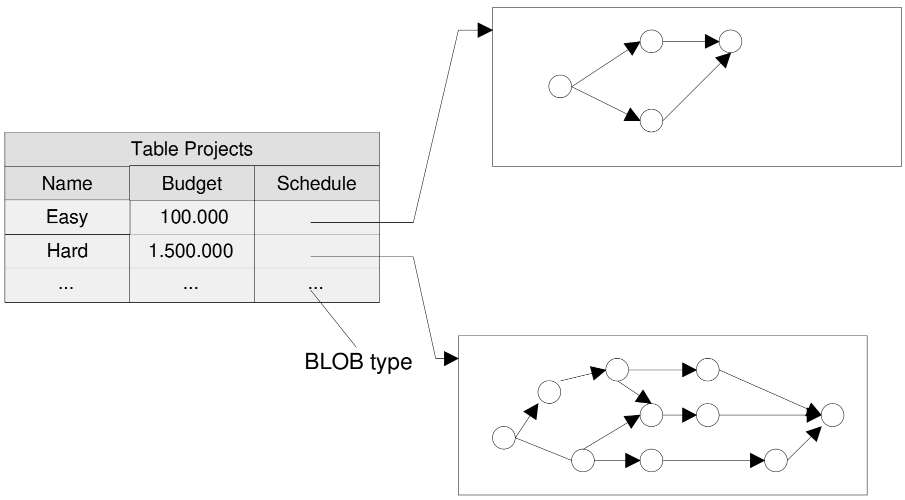



Figure 3: Coexistence of Object Data and Relational Data (picture adapted from [Rei+94])

### Related Patterns

Used in a pure form, the pattern is similar to the One Inheritance Tree One Table mapping. See also Representing Inheritance in a Relational Database [Bro+96].

## Patterns for Mapping Object Associations to Tables

This section presents two patterns used to map associations between objects: Foreign Key Association and Association Table. Note that the problem behind mapping 1:n associations is identical to Representing Collections in a Relational Database [Bro+96].

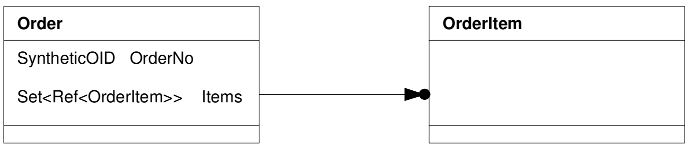
### Pattern: Foreign Key Association

#### Abstract

The pattern shows how to map 1:n associations between objects to relational tables

#### Example

Consider the classic Order / OrderItem example. A valid Order may have from zero to many OrderItems.




#### Problem

How do you map an 1:n association to relational tables?

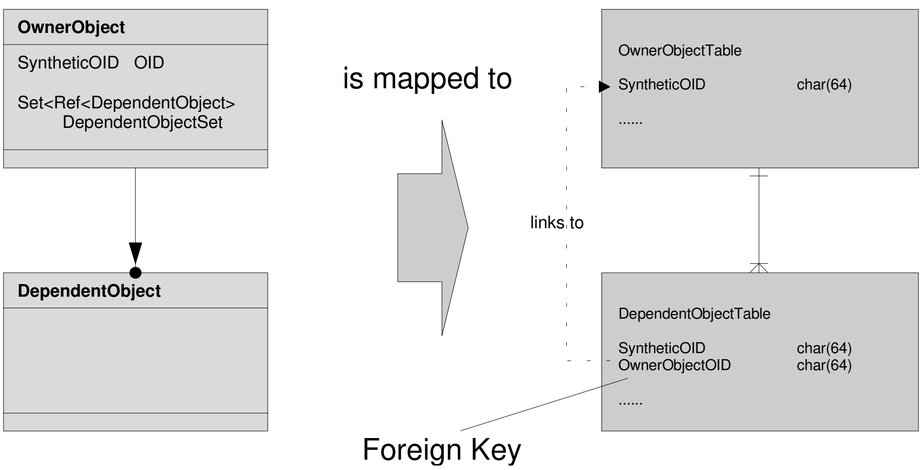
#### Forces

See the General Forces on page 1.

#### Solution

Insert the owner object's OID into the dependent objects table. The OID may be represented by a database key or a Synthetic Object Identity.

#### Structure



**Consequences**

- *Read performance*: Reading an Order object costs a join operation or two read operations - one of them multiple. You then have the Order object plus a set of references to all OrderItems.

- *Write Performance:* Writing all owned objects in an 1:n association using the pattern costs the number of changed owned objects as you would not write unchanged objects.

- *Performance and redundancy versus maintenance cost and normal forms*: The mapping scheme is the usual mapping scheme in relational database applications. It does not collide with normal forms. Hence it allows reasonable maintenance cost.

- *Space consumption* is near optimal - except the space required for the foreign key column in the DependentObject's table.

- *Ad-hoc queries*: As the mapping is common in relational database applications, ad-hoc queries are not harder or easier to write than in any relational database application.

- *Application style*: The mapping best suits relational style applications. You should not try to use a relational database for CAD or CASE applications. The reason for this lies exactly in mapping of associations to relational tables via foreign keys. Resolving an association costs a join operation or a second database access. Page based storage systems, such as OODBMS can handle similar use cases much faster but are worse at tuple processing - a stronghold of the relational model.

- *Integration of legacy systems*: As most relational legacy systems use exactly the mapping described, converting 1:n associations to objects is no source of new problems.

**Implementation**

- *General Performance:* If performance turns out insufficient you might use a Relational Database Access Layer below the object/relational mapping and apply performance improvement patterns such as Controlled Redundancy, Denormalization, or Overflow Tables [Kel+97]. This allows you to later optimize physical table schemes without affecting the logical mapping scheme proposed by this pattern.

- *Update performance*: When updating OrderItems (dependent objects) you should update only those that really have been changed and not each and every OrderItem that you did load into your working storage. Update and insert operations are expensive.

- *Prefetching dependent objects:* In case you know in advance, that you need access to all dependent objects (like OrderItems) for most use cases, you can get all data using a join operation and construct the owner object and the dependent objects from the result of a single database operation like:

```
select * from Order O, OrderItem I
 where   O.key = 'YourOrderKey' and
         O.key = I.OrderKey
```

This is faster than filling a container of Smart Pointers (`Set<Ref<OrderItem>>`) with object identities that have to be dereferenced one by one. For an Order with 20 OrderItems this costs 1 database access for the Order object, 1 multiple read access to get 20 object identities of the dependent objects plus 20 single record read accesses to get the OrderItem one by one.

### Related Patterns

In practice an 1:n association is often hard to distinguish from an aggregation. Therefore also consider the aggregation patterns <u>Single Table Aggregation</u> and <u>Foreign Key Aggregation</u>. The later is the same solution for a slightly different problem.

As an alternative to using foreign keys, consider <u>Controlled Redundancy</u>, <u>Denormalization</u>, and <u>Overflow Tables</u> [Kel+97].

The pattern is a close relative of <u>Mapping n:m Associations using Association Tables</u>. See also <u>Representing Object Relationships as Tables</u> [Bro+96].

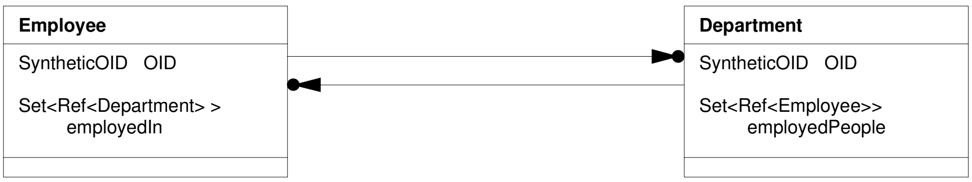
## Pattern: Association Table

### Abstract

The pattern shows how to map n:m associations between objects to relational tables

### Example

As an example we use the n:m association between an Employee object type and a Department object type. An Employee can work for more than one department. A department usually comprises more than one Employee.




### Problem

How do you map n:m associations to relational tables?

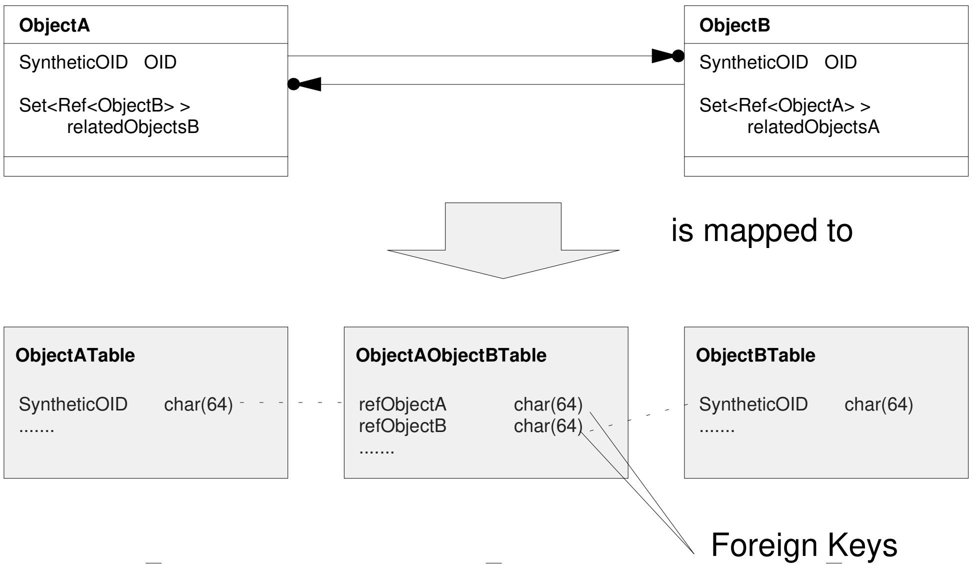
### Forces

See the General Forces on page 1.

### Solution

Create a separate table containing the Object Identifiers (or Foreign Keys) of the two object types participating in the association. Map the rest of the two object types to tables using any other suitable mapping patterns presented in this paper.

**Structure**




**Consequences**

The consequences are analogous to Foreign Key Association only adapted to the slightly different context. Hence we do not repeat them here.

**Implementation**

- *General Performance:* If performance turns out insufficient you might use database optimization patterns like Controlled Redundancy, Denormalization, or Overflow Tables [Kel+97]. In our case of a symmetric n:m association things get slightly more complicated as you have to break symmetry in order to apply performance optimizations.

- *Prefetching objects:* In case you know in advance, that you need access to all dependent objects (like Employees of a Department) for most use cases, get all data using a join operation and construct the Department object and the Employee objects from the result of a single database operation like:

```
select * from DepartmentTable D, EmployeeDepartmentTable ED,
              EmployeeTable E
 where   D.SyntheticOID = 'YourDepartment'
         D.SyntheticOID = ED.DepartmentKey and
         ED.EmployeeKey = E.SyntheticOID
```

  This is faster than filling a container of Smart Pointers (`Set<Ref<Employee>>`) with object identities that have to be dereferenced one by one. The same discussion with slightly different arguments could be found in Foreign Key Association.

#### Related Patterns

The pattern is closely related to Foreign Key Association. See also Representing Object Relationships as Tables [Bro+96].

## Known Uses

Single Table Aggregation, Foreign Key Aggregation, Foreign Key Association, and Association Table are used as an option in Persistence [www.persistence.com] or in the TopLink Smalltalk Framework [www.objectpeople.com/toplink/] and in most other persistence frameworks. The patterns are also used in an object/relational access layer by HYPO-Bank [Col+96,Kel+96] and in an object/relational gateway by POET [POE96].

One Inheritance Tree One Table and One Class One Table are discussed as a design option in a concept for an object/relational gateway by POET [POE96] together with One Inheritance Path One Table. The later pattern has been used by the Champs and HYPO projects [Col+96, Hah+95, Kel+96].

Objects in BLOBs has been used in the SMRC research prototype [Rei+94, Rei+96].

## Credits and Acknowledgments

Kyle Brown and Bruce Whitenack discuss Representing Objects As Tables and many other patterns we cited above in their "Crossing Chasms" pattern language [Bro+96]. Stanislav Kumsta and Uwe Steinmüller have written a series of short patterns on mapping objects to tables [Ste97]. Wolfgang Hahn, Fridtjof Toenniessen, and Andreas Wittkowski describe some similar experiences in their report on the Champs project [Hah+95]. And here the circle closes as Uwe Steinmüller has also been a member of the Champs team. The same problems have again been discussed by a team working on the POETGate object/relational gateway for the POET object database [POE96] and finally we have made own experience with the HYPO project [Bar+95]. The list could be continued. I'd like to express my thanks to the two reviewers so far: Kyle Brown and Jens Coldewey.

## References

[Bar+95] **Christian Barschow, Petra Hieber, Wolfgang Keller, Christian Mitterbauer**: *Persistente Objekt unter Berücksichtigung bestehender relationaler Datenbanken*, Internal Technical Report, HYPO Bank, München 1995.

[Bro+96] **Kyle Brown, Bruce G. Whitenack**: *Crossing Chasms, A Pattern Language for Object-RDBMS Integration*, White Paper, Knowledge Systems Corp. 1995. A shortened version is contained in: **John M. Vlissides, James O. Coplien, and Norman L. Kerth (Eds.):** *Pattern Languages of Program Design 2*, Addison-Wesley 1996.

[Col+96] **Jens Coldewey, Wolfgang Keller:** *Objektorientierte Datenintegration - ein Migrationsweg zur Objekttechnologie*, Objektspektrum Juli/August 1996, pp. 20-28.

[Col96] **Jens Coldewey:** *Decoupling of Object-Oriented Systems - A Collection of Patterns*; sd&m GmbH & Co.KG, Munich, 1996; available via http://www.sdm.de/g/arcus/

[Dat95] **C. J. Date:** *An Introduction to Database Systems, Sixth Edition*; Addison-Wesley 1995.

[Hah+95] **Wolfgang Hahn, Fridtjof Toennissen, Andreas Wittkowski**: *Eine objektorientierte* Zugriffsschicht *zu relationalen Datenbanken*, Informatik Spektrum 18(Heft 3/1995); pp. 143-151, Springer Verlag 1995

[Kär95] **Winfried Kärtner**: *Konzept: Datentypen in der Hypo-Bank*, Internal Technical Paper, HYPO Bank and sd&m, 1995.

[Kel+96] **Wolfgang Keller, Christian Mitterbauer, Klaus Wagner**: *Objektorientierte Datenintegration über mehrere Technologiegenerationen*, Proceedings ONLINE, Kongress VI, Hamburg 1996.

[Kel+97] **Wolfgang Keller, Jens Coldewey**: *Relational Database Access Layers: A Pattern Language*, in „Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences„, Washington University, Department of Computer Science, Technical Report WUCS 97-07, February 1997.

[Loh+91] **Guy M. Lohman, Bruce G. Lindsay, Hamid Pirahesh, K. Bernhard Schiefer**: *Extensions to Starburst: Objects, Types, Functions, and Rules*. CACM 34(10) pages 94-109 (1991)

[POE96] **POET GmbH**; *POETGate - ein Konzept zur Integration relationaler Daten in die POET-Architektur*; Poet GmbH; 1996.

[Rei+94] **Berthold Reinwald, Stefan Deßloch, Michael J. Carey, Tobin J. Lehman, Hamid Pirahesh, V. Srinivasan:** *Making Real Data Persistent: Initial Experiences with SMRC*. POS 1994: 202-216

[Rei+96] **B. Reinwald, T. J. Lehman, H. Pirahesh, V. Gottemukkala**: Storing and using objects in a relational database; IBM Systems Journal, Vol. 35, No. 2, 1996.

[Rum+91] **James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen:** *Object-Oriented Modelling and Design*, Prentice Hall, 1991.

[Ste97] **Uwe Steinmüller**, personal communications, 1996, 1997.

[You+95] **Ed Yourdon, Katharine Whitehead, Jim Thomann, Karin Oppel, Paul Nevermann**: *Mainstream Objects, An Analysis and Design Approach for Business*; Prentice Hall 1995.