Object/Relational Access Layers

A Roadmap, Missing Links and More Patterns

Wolfgang Keller
Liebigstr. 3, 82166 Lochham, Germany
Email: wk@objectarchitects.de
http://www.objectarchitects.de/

Abstract

Designing software to connect an object-oriented business system with a relational database is a tedious task. Object-orientation and the relational paradigm differ quite a bit. An application that maps between the two paradigms needs to be designed with respect to performance, maintainability and cost to name just a few requirements. Luckily there are numerous patterns of object/relational access layers, but looking at the body of pattern literature you will find that some patterns are still to be mined, while there's no generative "one stop" pattern language for the problem domain. This paper provides a systematic roadmap of the patterns in the field, and fills some pot holes on the road towards a full pattern language for object/relational access layers by providing some missing patterns and links.

Introduction

Most large scale business systems follow a three layer architecture. They provide a user interface layer on top of a business object layer. The business objects need to be made persistent somehow in a persistence layer.

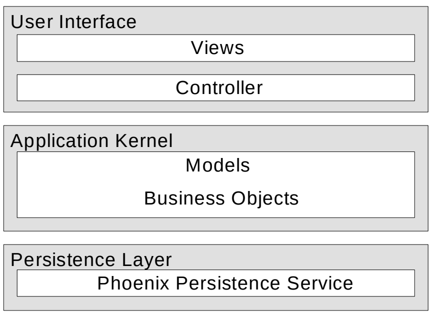


Figure 1: Structure of Layered Architecture for Business Systems.

If you want to use pure object-orientation to implement your business system you have to decide which database paradigm to use for your system. Today you have a choice of using:

- object-oriented database systems (OODBMS),
- object/relational access layers on top of a relational database,
- or a relational database access layer, which will lead to a so called representational business object layer. See [Kel+98a].
- object/relational databases plus an access layer. We will not write about these kinds of applications as we don't have any practical experience with this kind of databases.

The functionality covered by the first three of the above options is depicted in Figure 2.

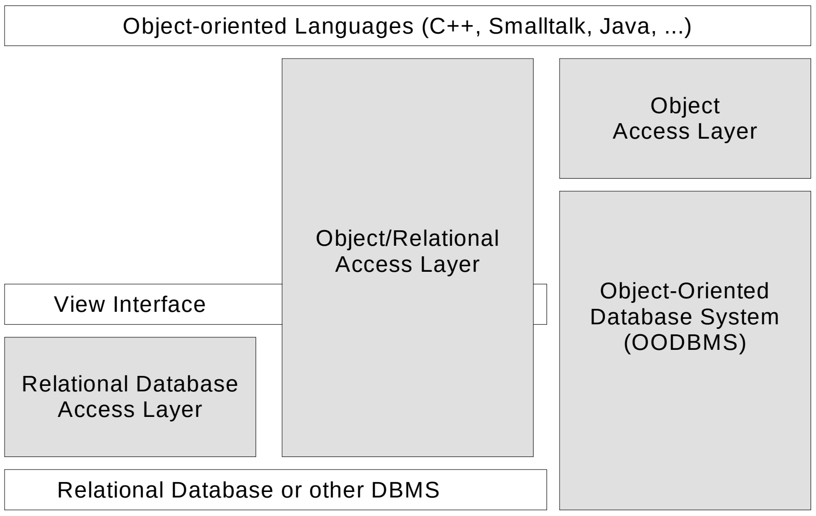

Object-oriented Languages (C++, Smalltalk, Java, ...)
Object Access Layer
Object/Relational Access Layer
View Interface
Object-Oriented Database System (OODBMS)
Relational Database Access Layer
Relational Database or other DBMS

Figure 2: Three Different Kinds of Database Access Layers

This paper contains patterns or references to patterns that will help you design or understand object/relational access layers. The paper fills some pot holes on the road towards a full generative pattern language for object/relational access layers, and provides some missing patterns, links to existing patterns and names the rest of patterns that have yet to be mined.

Roadmap of the Pattern Language

A roadmap structures a pattern language. For object/relational access layers we draw a roadmap according to the fields of decisions you have to make when you are designing and using an object/relational access layer.

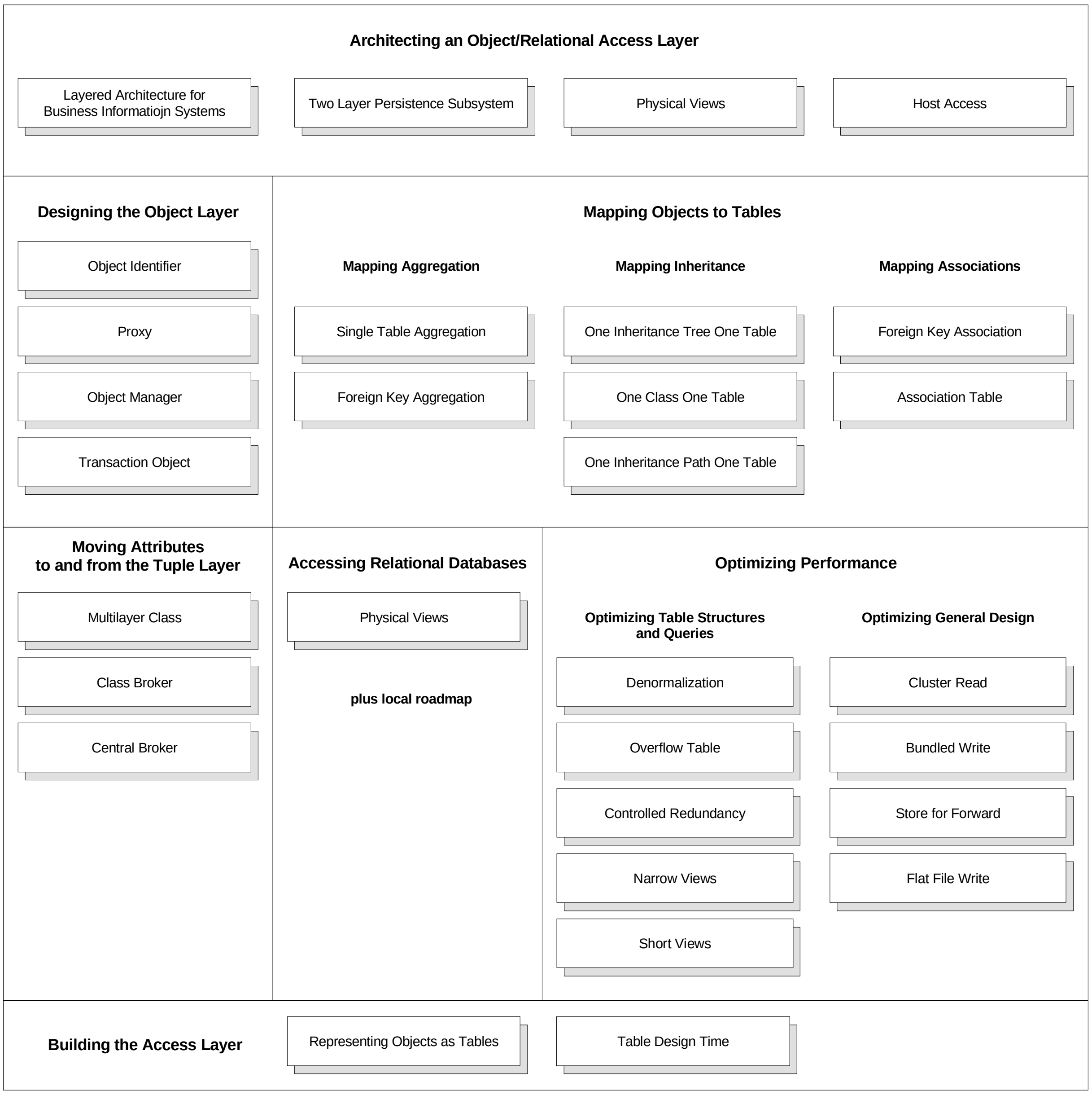


Figure 3: A Roadmap of Object/Relational Access Layer Patterns

Architecting an Object/Relational Access Layer[1] describes how to structure an object/relational persistence subsystem in the global context of a layered architecture for business systems. The architecture consists of two layers: an object layer that contains the infrastructure to persist business objects and a tuple layer that encapsulates a relational database. Hence there are patterns for **Designing the Object Layer** and **Accessing Relational**

[1] Some notational conventions: **Groups of Patterns** are marked as bold face. <u>Single patterns</u> are marked as underlined.

Databases. The next task is **Mapping Objects to Tables.** You also have to come up with a design for **Moving Attributes to and from the Tuple Layer**.

Having worked your way through the above patterns you will usually find that the performance of the system needs to be improved–- see the patterns for **Optimizing Performance**. And finally **Building the Access Layer** will tell you how to organize your tasks when using an object/relational access layer.

Forces Driving the Language

The following set of forces is adapted from *Accessing Relational Databases* [Kel+98a]. There are only a few more forces here than for relational access layers:

- *Functionality versus cost:* Besides the mandatory features of object-oriented database systems, listed below there are also a lot of optional features. Even some of the mandatory features are known to be expensive to implement, and some of the optional ones are even harder. You should therefore balance the features you'd like to incorporate into an object/relational access layer with the budget your users are willing to spend.

- *Separation of concerns versus cost:* Database programming is complex and so are object-oriented programming languages. Mapping one concept to the other will add up to more than just the combined complexity. The easiest way is to separate the application programming from the database programming and to separate the object-oriented database aspects from the relational database aspects. You are then able to exploit well-known patterns for each of the problem domains. The cost of separated layers has to pay off with increased maintainability and easier performance tuning.

- *Performance:* Database tuning, locking strategies and clever caching are crucial to achieve acceptable performance of a business information system. Since a database is several orders of magnitude slower than the main processor running the OO language you map, tuning will concentrate on database access. Tuning is an iterative process. To optimize database access you may change the access layer architecture and behavior, the physical parameters of the storage system, as well as the table layout, or the API to access the database.

- *Flexibility versus complexity:* Since database tuning is crucial, you want to have an encapsulation of the database that allows frequent changes to the underlying data model while your upper layers of software (the application kernel and most of the access layer) remain untouched. Unfortunately, the more flexible a system is, the more complex it will be.

- *Legacy systems:* You seldom develop business information systems from scratch. Instead, you have to connect to legacy systems which you are not allowed to touch. Usually you can not supersede the complete legacy code, because big bang strategies are risky and expensive, but, the structure of legacy data rarely fits your needs - if it has any structure. You may also have to bridge several generations of database technologies. To keep your application maintainable you have to encapsulate the legacy access. This is a particular strong force during reengineering projects.

- *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 a 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 examples 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.

Architecting an Object/Relational Access Layer

List of Requirements

Given that you have to use a relational database and given that you want a fully object-oriented application kernel it is good to have a list of typical functionality for an object-oriented database. The Object-Oriented Database System Manifesto [Atk+89] contains a very comprehensive list of the functionality you might want to provide (see Table 1) with your object/relational access layer.

OODMS Manifesto: **Mandatory Features**	**Object/Relational Access Layers:** **Covered by**
(1) Complex Objects	Your programming language for business objects (like C++, Smalltalk, Java, ...), your RDBMS plus an access layer.
(2) Object Identity	See Object Identity Pattern
(3) Encapsulation	Your programming language
(4) Types and Classes	Your programming language
(5) Class or Type Hierarchies	Your programming language plus patterns for **Mapping Objects to Tables**.
(6) Overriding, overloading and late binding	Your programming language

OODMS Manifesto: Mandatory Features	**Object/Relational Access Layers: Covered by**
(7) Computational Completeness	Your programming language
(8) Extensibility	Your programming language plus patterns for **Mapping Objects to Tables**.
(9) Persistence	Whole access layer plus relational database (RDBMS).
(10) Secondary storage management	RDBMS
(11) Concurrency	RDBMS plus patterns for transaction control and locking strategies.
(12) Recovery	RDBMS
(13) Ad Hoc Query Facility	access layer on top of RDBMS

Table 1: Core responsibilities of an Object-oriented Database Management System

Most of the functionality listed in Table 1 comes with your object-oriented programming language (like 1, 3, 4, 5, 6, 7, 8). The challenge is to make your object-oriented programming language’s objects persistent, giving them the ability to survive the termination of the actual process and to be used again in other (also in parallel) processes.

Therefore the other requirements are typical requirements that you find for databases (like 9, 10, 11, 12,13). See any database book for an explanation, e.g. [Dat95].

Forces Driving the Architecture

The forces driving the architecture are naturally the ones that drive the language as the architecture represents the top level design decisions of an object/relational access layer.

Local Roadmap

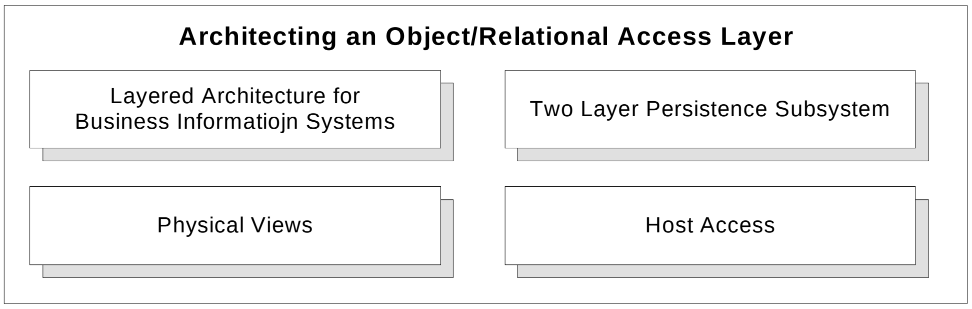

Architecting an Object/Relational Access Layer
Layered Architecture for Business Informatiojn Systems
Two Layer Persistence Subsystem
Physical Views
Host Access

Figure 4: Local Roadmap: Architecting an Object/Relational Access Layer

Pattern List

- Layered Architecture for Business Systems: What is a good overall architecture for Business Systems? See [Ren+97] for a discussion in pattern form and [Bus+96] for a general discussion of layered architectures (Layers).
- Two Layer Persistence Subsystem: What is a good structure for a persistence subsystem?
- Physical Views: How do you provide an easy to use interface to your physical database tables? See [Kel+98a] and also the Query Objects Pattern in [Bra+96].
- Host Access: How do you link you database access layer to a transaction based host database server?

Patterns

Pattern: Two Layer Persistency Subsystem

Problem

What is good way to structure an object-oriented database or an object/relational access layer?

Forces

Remember the above discussion on *Separation of concerns versus cost:* Database programming is complex, storage subsystems are complex but they are known abstractions. Object-oriented programming languages are also proven concepts. Both have enough complexity. Mapping one concept to the other and not dividing into further subsystems could easily sum up to a nightmare of complexity. The easiest way is to separate the concepts of object-orientation from those of database programming and to separate the object-oriented database aspects from the relational database aspects. You are then able to exploit well-known patterns for each of the problem domains. The cost of separated layers has to pay off with increased maintainability and easier performance tuning.

Application style is another force. You should be able to adapt you object persistence subsystem to the different application styles mentioned above. It makes a great difference, whether you intend to write a transaction oriented system or a system that can best be described with *check in/check out* persistence.

Finally the possible *integration of legacy data sources* will have its effects on you design.

Solution

Build your system as two subsystems that form a layered structure. The upper layer, called the object layer, encapsulates the concepts of object-orientation while the lower layer, called the storage manager, offers a high level interface on top of your physical storage devices or file system. A relational database in this context is just another physical storage device.

Structure

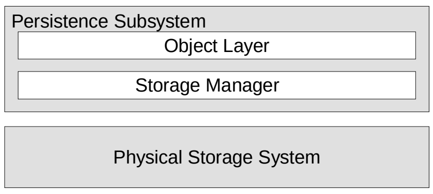


Figure 5: Two Layer Structure of a Persistency Subsystem for Object-Oriented Programming Languages (OOPLs)

Assign the following responsibilities from *The Object-Oriented Database System Manifesto* [Atk+89] to the layers.

Object Layer:

The object layer encapsulates the concepts of object orientation. It has the following responsibilities: (1) Complex Objects, (2) Object Identity, (3) Encapsulation, (4) Types and Classes, (5) Class or Type Hierarchies, (6) Overriding, overloading and late binding, (7) Computational Completeness, (8) Extensibility, (13) Ad Hoc Query Facility. This is the object-oriented programming languages part of the requirements listed in Table 1.

Storage Manager:

The Storage Manager provides an interface to a Physical Storage Subsystem. It has the following responsibilities: (9) Persistence, (10) Secondary storage management, (11) Concurrency, (12) Recovery, (13) Ad Hoc Query Facility. This is the database part of the requirements listed in Table 1. The only exception is the “*Ad Hoc Query Facility*”. The Ad Hoc Query Facility is a database concept that you wrap at the level of your object-oriented language in order to offer your user the equivalent of SQL. Therefore you have to deal with some form of Object SQL (also called Object Query Language (OQL) [ODMG93]) in both layers.

This discussion could lead to some form of abstract pattern. Whenever you have two paradigms that need to be mapped on one another, you can come up with an architecture that consists of two layers. These layers contain the respective abstractions of the two paradigms, and the upper layer (the paradigm you want to map onto another) needs some code to call the lower layer – this code is mostly in the broker patterns (see **Moving Attributes to and from the Tuple Layer**)

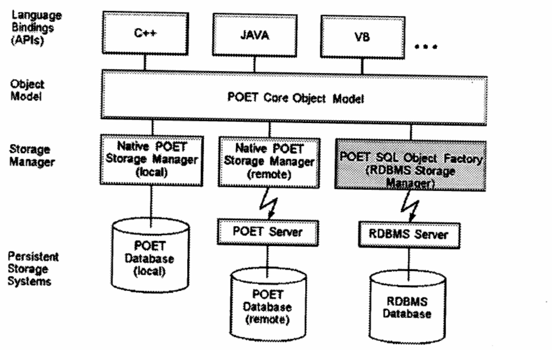
Consequences

Manageability and complexity: This approach breaks the problem down into manageable parts by cutting it into two halves – and one of these, the storage manager, is not a new problem but a kind of component with a long lasting design history.

Application Style: You can adapt your persistence subsystem to different application styles by plugging in different storage managers. The need to adapt to *transactional legacy systems* will influence your storage manager but not your object layer.

Variants

An object/relational access layer is a variant of an object-oriented database. An architectural sketch from POET makes this quite evident. POET is an object-oriented database that uses a relational database (plus an access layer) as its storage subsystem.



If you do not use an object-oriented database with a relational database as its storage manager you have to build an object/relational access layer. For the rest of the paper we will use the term tuple layer instead of storage subsystem as we use relational databases to store our objects.

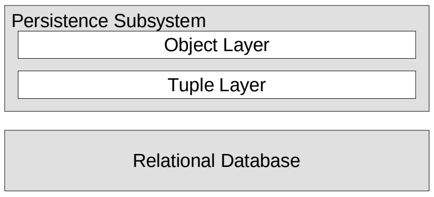


Figure 6: Two Layer Structure for an Object/relational access Layer

Related Patterns

We have used the concepts of Layers [Bus+96] here. All the other patterns in this paper are further solutions to the problems of how to build such an access layer.

Known Uses

Most object-oriented databases and object/relational access layers are built this way. We have already cited POET [POE97] as an arbitrary example. TopLink is an example that uses the pattern in a object/relational access layer product [TOP97a,TOP97b]. There are many project solutions, that follow the same architecture [Bar+95, Hah+95, Kel+98b, Sta+97, Wal+95]. Another use of the architecture can be found in [Hei98]. Heinckiens distinguishes an object layer, a database layer and brokers between the two layers, which he calls Impedance Mismatch Resolvers.

Pattern: Host Access

Example

You have to build an object/relational access layer alongside legacy applications on a host computer. Both suites of applications, the old transaction based applications and your new object-oriented applications should use the same host database access layer so that you have single source on you host computer. Most off the shelf access layer products are constructed on top of an ODBC interface. This does not combine well with a transaction system, as running a host as a remote SQL server is not the way things are handled.

Problem

How do you connect an object/relational access layer to a host computer running a transaction system?

Forces

Performance versus straightforward design: The straightforward design that provides access to a relational database on a host computer is to run the host as a remote SQL server. Unfortunately this is not fast enough and does not offer enough possibilities for tuning on a transaction system.

Single source: You want to use access layer modules from your host applications as well as from client applications.

Integration of legacy systems: You might want to add other legacy data sources like IMS/DB databases or flat files.

Solution

Write all queries to a communication agent, using bundled write. Install another communication agent on your host computer that unpacks the query packets and executes them one by one under the control of the host transaction monitor. Send back a packet containing query results or the return codes of the access layer modules from the host computer.

Structure

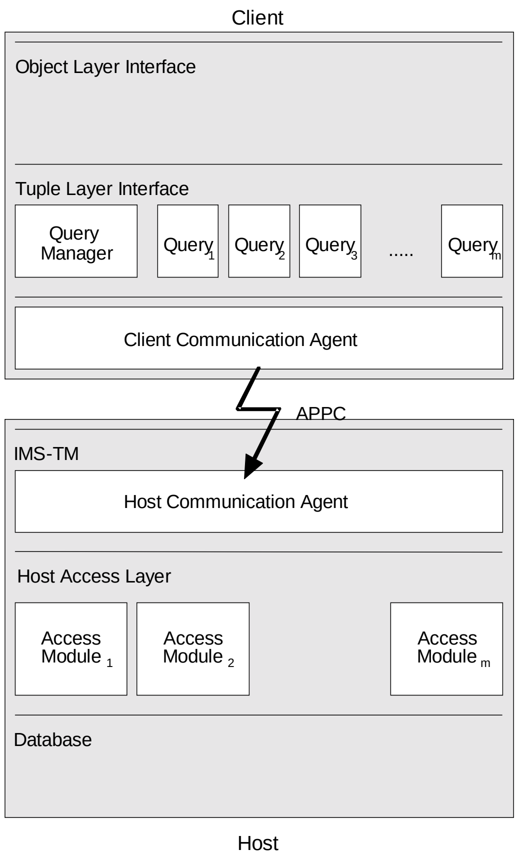


Figure 7: Connecting and object/relational access layer to a host transaction system [Kel+98b]

The structure (see Figure 7) shows the following similarities and differences compared to a normal object/relational access layer based on a remote database driver based on ODBC or similar:

- The interface of the tuple layer remains unchanged.

- Insert a client and a host communication agent below the tuple layer. The client communication agent bundles write queries (see bundle manager in the bundled write pattern). The client communication agent is an object that buffers requests and does not execute them before it is told to do so. If the other communication agent on the host side receives a bundle of requests, it executes them one by one by calling access modules, buffers the results and sends back a bundle of results. The client communication agent only checks return codes and delivers results if necessary.

- Install another communication agent as a dispatcher and write it as a host transaction (e.g. under IMS/TM or CICS).

- Have this host communication agent call the host access layer modules that implement the functionality of your client's query objects - the query objects are proxies for the host access layer modules.

- The host access layer modules will access the database.

Consequences

Performance versus straightforward design: This solution offers reasonable performance, as we can see in more than one independent productive systems [Bar+95, Sta+97]

Single source: is given, as the host database modules can be used from both object-oriented client applications and conventional host applications. A project to write the host access layer can normally justified from the gains of productivity that result from using the host access layer from host applications alone.

Integration of legacy systems: It is straightforward to wrap another data source than a relational database by host access layer modules.

Related Patterns

This pattern is an application of proxies [GOF95] in the sense that the query objects on the client are proxies for the host access layer modules. The communication agents on the host and on the client implement bundled write.

Known Uses

The Hypo-Project [Bar+95, Kel+98b] uses this pattern as well as the Phoenix project. TopLink offers a separate *mainframe interface* as a byproduct to its standard access layer product. This is used in the Phoenix project [Sta+97] together with a host access layer written in C.

Designing the Object Layer

Forces Driving the Design of the Object Layer

The main force driving the design of the object layer are *features versus cost*: You can come up with many expensive features like complex queries, nested parallel transactions and so on, but implementing them does not come cheap.

Local Roadmap

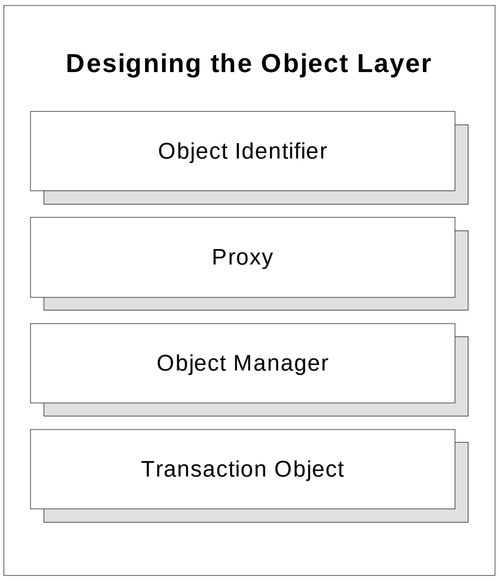


Figure 8: Local Roadmap for Designing the Object Layer

Pattern List

The patterns you need to construct the object layer have all been described in other papers.

- Object Identifier: How do you represent an object's individuality in a relational database? See [Bro+96]. Some would doubt today that this is a pattern. The concept is described very clearly in The Object-Oriented Database System Manifesto [Atk+89], which gives a definition of Object Identity in the context of object-oriented databases. The same definition is applicable for object/relational access layers.

- Proxy: How do you prevent all related objects to be loaded whenever you touch one object that has relations to many others? See [GOF95], the unofficial version of "Crossing Chasms" [Bro+96] and also Scott Meyers on Smart Pointers [Mey96]. See also the Reference class in [Hei98, Section 7.5].

- Object Manager: How do you preserve object identity? See the View Cache pattern in [Kel+98a], or the Object Manager in [You+95, pages 291-292, Max96] plus the unofficial version of "Crossing Chasms" [Bro+96]. To understand the interactions between the Object Manager, Transaction Objects and the objects of the object/relational access layer, replacing the term Object Manager with View Cache, and the term object with Logical

View.

- Transaction Object: How do you handle transactions at a user code level? For a solution see the pattern in Accessing Relational Databases [Kel+97] which was in fact adapted from what we saw in object/relational access layers and the ODMG standard [ODMG93] or see [Hei98, Chapter 10]

- Database Object Protocol: How do you provide a uniform protocol for all your persistent objects? You derive them from a DatabaseObject. This is an application of abstract base classes.

- Narrow Views and Short Views [Kel+98a]: are two patterns that should be considered, when designing the ad hoc query capabilities of your access layer.

- Basic Relationship Patterns [Nob97] plus the ODMG-Standard [ODMG93] show you how to implement object relationships. Mapping interobject relationships to relational databases is treated in **Mapping Objects to Tables**.

Accessing Relational Databases

Forces Driving the Design of the Tuple Layer

As **Mapping Objects to Tables** is treated in a separate fragment of the language, the remaining field of decisions for the tuple layer is the design of the query interface - The dominant forces here are *Ease-of-use versus power of the interface:* Your interface should be easy to use. On the other hand the complexity of a database interface stems from its power. Hence, the interface of the databases encapsulation should be easy to use but still powerful enough for your project. In object/relational access layers you can live with moderate complexity as you have another layer upon your queries - your persistent objects. You should only be forced to use SQL database queries directly in very rare cases.

Local Roadmap

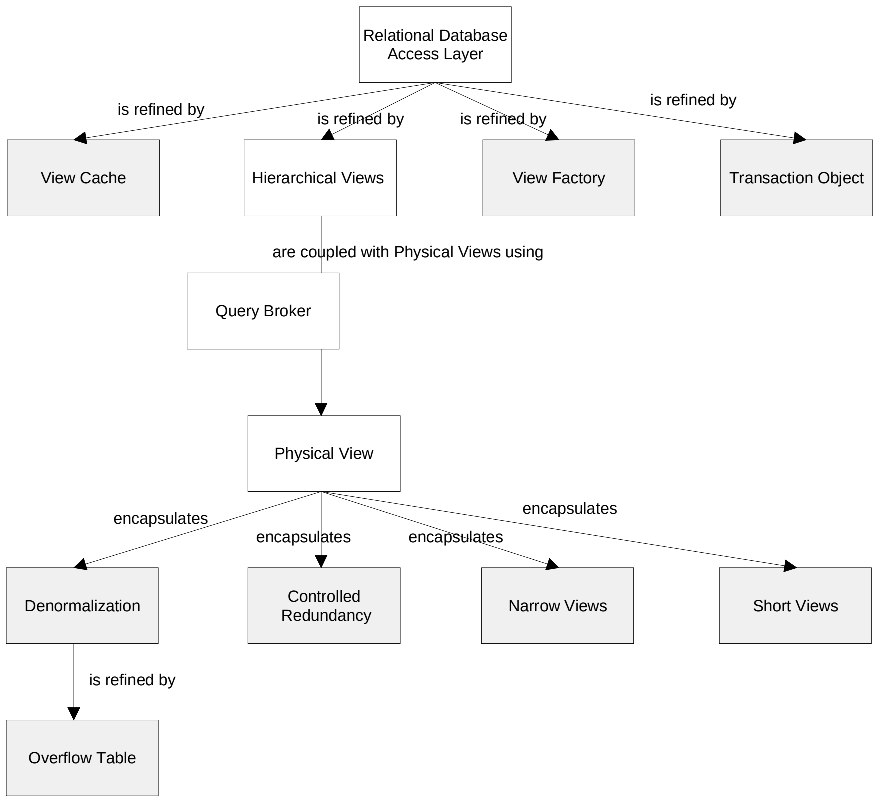


Figure 9: Roadmap from Accessing Relational Databases [Kel+98a]. Merely all patterns are also found in object/relational access layers.

If you take a fully fledged relational access layer that can also be used as a surrogate for an object/relational access layer its interesting to see where the patterns used there are moving to in this language.

- The Object Manager in this pattern language is the equivalent of the View Cache. So the View Cache is moving one layer up.
- Your programming language's persistent objects are analogous to Hierarchical Views.
- You will have a persistent object factory - an equivalent of the View Factory in relational database access layers.
- The Transaction Object exists verbatim but is moved one layer up to the object layer. See [Kel+97] and [Kel+98b].
- The Query Broker in relational database access layers is substituted by the patterns you need to move **Attributes to and from the Tuple Layer.**
- Physical Views are the core abstraction of the tuple Layer.

- The performance optimization patterns below Physical Views can also be used in object/relational access layers - they are complemented by the patterns for **Mapping Objects to Tables** and some more patterns for **Optimizing Performance**

Pattern List

So the pattern that remains in the tuple layer is Physical Views which is also known as Query [Bra+96]. If you want real luxury, you can also add an additional layer of Logical Views to implement Cluster Read

Moving Attributes to and from the Tuple Layer

There is a set of patterns dealing with the question of how to move attribute values across the border between the two layers of an object/relational access layer –- the object layer and the tuple layer.

Forces Driving the Way you Move Attributes

The way you will move attributes from objects to queries in the tuple layer and from the tuple layer to your objects attributes is influenced by the programming language you use. In C++ private variables are private, and unlike Smalltalk, there are no `>>instVarAt` methods to get hold of private instance

Therefore C++ techniques have to be based on code generation or hand written methods in the object layer (that is, methods which the persistent objects need to implement). This is called a push down approach, because the objects are pushing their content to a lower layer. Smalltalk offers rich possibilities to get information out of objects regardless whether it is public or private, so you can economize on code quantity (no code is good code) and write access layers that resemble a meta system (see the Reflection pattern [Bus+96]). A generic mapper can encapsulate all the mapping meta information and pull the objects' information down to the lower layer and stuffs it into queries it generated from the mapping meta information at run-time.

Local Roadmap

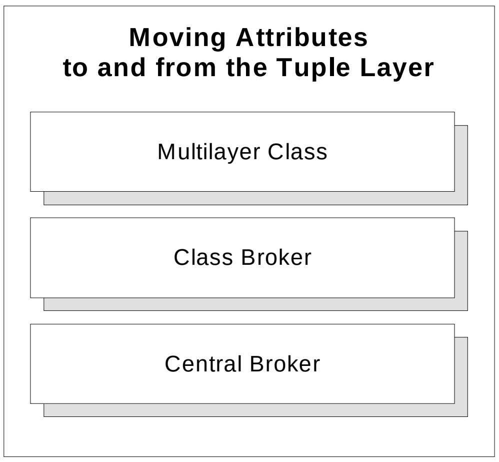


Figure 10: Local Roadmap for Moving Attributes to and from the Tuple Layer

Pattern List

- Multilayer Class: This pattern provides a solution to the question: How do you design the communication between the layers without extraordinary effort in C++? You do this by generating the code to access the database into separate methods. These methods use the tuple layer directly, without any further decoupling. This pattern Was rated a last resort pattern at PLoP96 [Col+96b]. A last resort pattern is one that is in broad use in absence of better solutions.

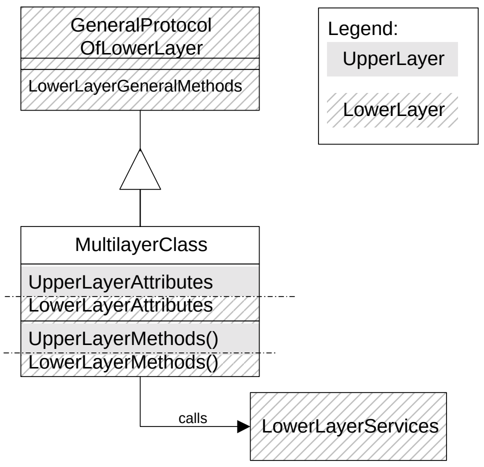


Figure 11: Structure of the Multilayer Class. Shaded parts denote methods and attributes of the upper layer (object layer). Dashed parts show lower layer (tuple layer) members

Your object's method protocol has a lower level protocol called database which contain everything needed to push down its attributes The pattern provides a proven and often used solution (for example in the POLAR Framework or the HYPO Framework [Bar+95,Kel+98b]) to do things the C++ way. Bobby Woolf, our smalltalking shepherd, was simply disgusted.

- Class Broker: Is a way to do things more the Smalltalk way by concentrating the mapping for a class in a separate Broker class (see the unofficial version of [Bro+96]). A similar pattern has been described as "Strong Layering" [Via+97]. Impedance Mismatch Resolvers are also a form of Class Brokers. See also [Hei98, Section 5.4.]

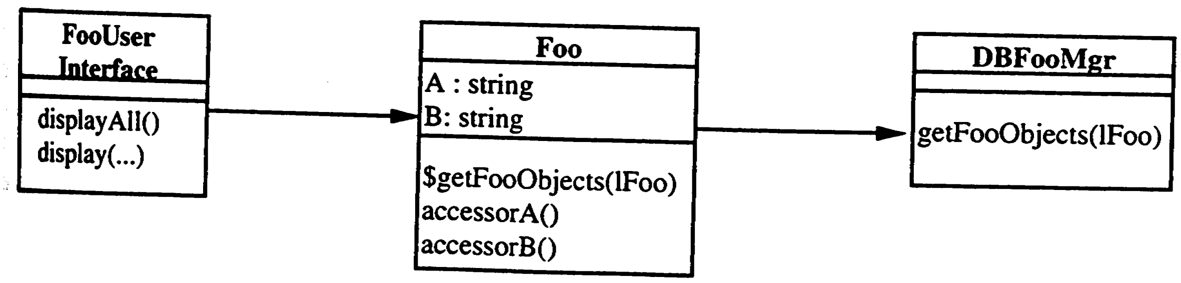


Figure 12: The DBFooMgr is a per Class Broker, which is called by the Foo Object to fetch some attributes. The DBFooMgr will call the tuple layer's Physical Views to obtain the data from the database [Via+97].

- Central Broker: Yet another way to do things in reflexive languages like Smalltalk is to build a central broker that evaluates classes' meta information, uses `>>instVarAt` to pull out the attributes and push them into query objects, generated from the mapping meta info. TopLink [TOP97a,TOP97b] uses this approach. The Central Broker is a Singleton [GOF95]

Mapping Objects to Tables

Mapping Objects To Tables [Kel97] is a stand alone pattern language fragment of it's own, that answers questions of how object-oriented constructs like inheritance, aggregation, or relations can be mapped to the semantics of relational databases.

Local Roadmap

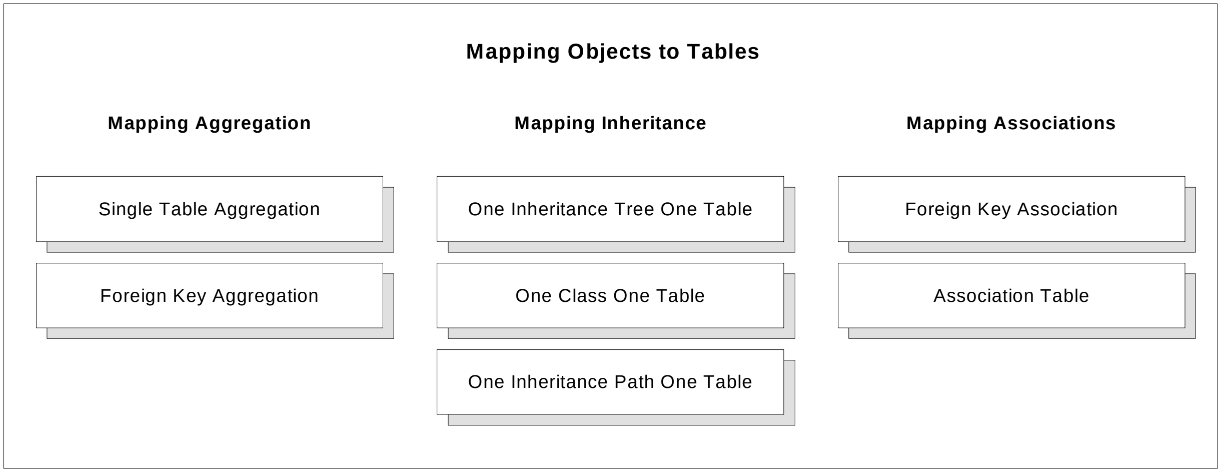


Figure 13: Local Roadmap for Mapping Objects to Tables

Forces Driving Mapping and Performance Optimization

The forces driving the mapping and performance optimization patterns have been in other papers on *Mapping Objects To Tables* [Bro+96, Kel97, Hei98] and *Accessing Relational Databases* [Kel+98a]. Space does not permit us to include these patterns here in full, so this section contains only a brief overview.

Performance is a major consideration in object/relational if you build an access layer. To be usable, the layer has to work fast enough. Often there are tradeoffs between *Read and write/update performance*. *Flexibility* and *maintenance cost* will in most cases conflict with *complexity*, so the more *flexible* you build a system the more *complex and expensive* it will become. *Performance* can often be improved by redundancy and will then collide with versus maintenance cost and normal forms of the relational model [Dat95]. *Space consumption* of the database also collides with *Performance* of an application.

The need for *query processing* is another force with some influence - it collides with *performance optimal mappings*. For example building a data warehouse often implies separating the queryable data from the data needed for fast online processing. And finally, potential *integration with legacy* system via the database will often collide with the best performing mapping and will add *complexity* if you have to integrate existing table structures.

Pattern List

- Single Table Aggregation and Foreign Key Association.: How do you map aggregation to relational tables?

- One Inheritance Tree One Table, One Class One Table, or One Inheritance Path One Table.: How do you map an inheritance hierarchy of classes to database tables?:

- Foreign Key Association.: How do you map an 1:n association to relational tables?:

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

- Objects in BLOBS: Solves all the above problems in one pattern.

Find all the above Patterns in [Kel+97] and also with another level of detail in [Bro+96].

Optimizing Performance

Once you have finished the first cut of your application you will almost always feel the need to improve performance. The general pattern for this is a bit too abstract to offer real help. Simply the fact that you would have a hard time to assign a name other than performance optimization to it is an indicator that this a general rule - and not a pattern.

Problem

How do you optimize performance in an application using a database?

Forces

The forces here are your wish for optimal performance on the one hand and the complexity and cost of an optimal access layer on the other hand. Other tradeoffs include memory usage (caching) versus use of slow I/O. See [Kel97] or [Kel+98a] for extensive lists.

Solution

Try to reduce database traffic and disk I/O to a minimum that still yields a maintainable application at reasonable cost

The above "solution" contains balancing of forces as the solution - it is therefore no ready solution. A deeper analysis of the factors that cause bad performance leads to a series of patterns that can be split into two categories.

- **Optimizing Table Structures and Queries:** Can be achieved by a series of performance patterns that you use to tune performance depending on your business objects' structure and access behavior. These patterns deal with optimizing table structures and access behavior. The result of applying them is usually lost for the next project.

- **Optimizing General Design:** Is a set of performance patterns that you incorporate in the static design and architecture of the access layer itself. These patterns deal with optimizing the access layer's structural design for performance If you take the layer to the next project, that kind of tuning will be already done.

Local Roadmap

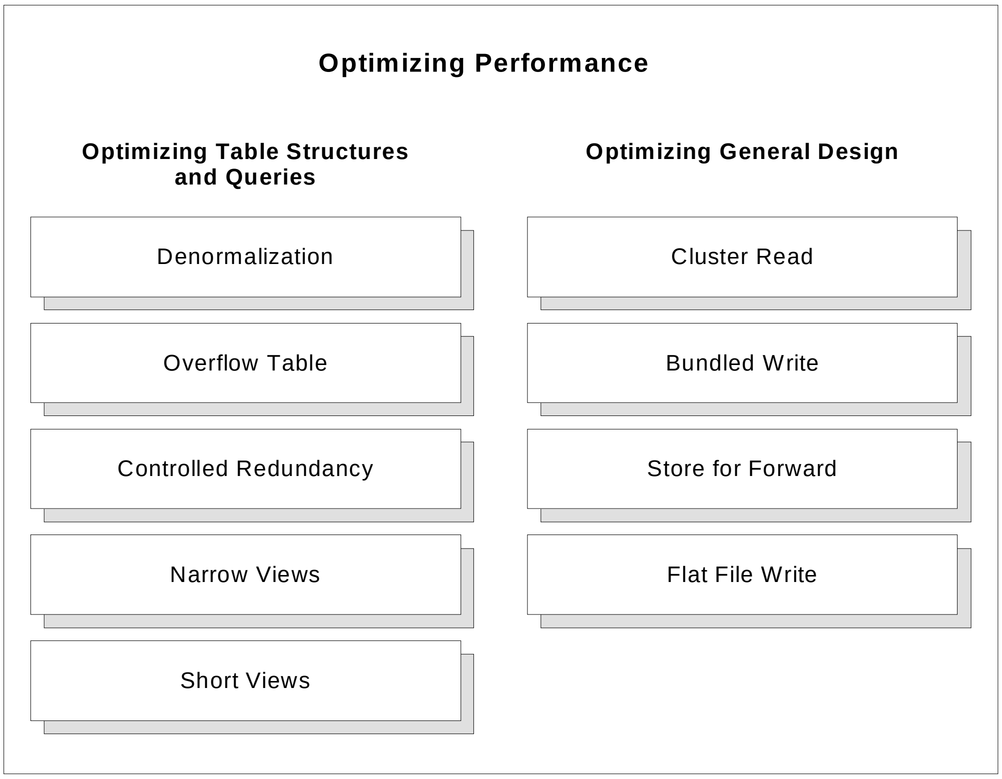


Figure 14: Local Roadmap for Optimizing Performance

Pattern List

The first category **Optimizing Table Structures and Queries** has been documented in [Kel+97]:

- Denormalization: How can you manage to read and write object clusters with a single page database access when you have a parent/child relation?

- Overflow Table: You have followed the Denormalization pattern’s advice and have denormalized a relation. What do you do with those objects that have more dependent objects than the number that you integrated into the parent object's table?

- Controlled Redundancy: How can you manage to read object clusters with a single page database access when you need to read data from a parent object's table?

- Narrow Views: What kind of database or object level views should you use for filling list boxes?
- Short Views: How do you speed up filling of list boxes and prevent unnecessary data from being loaded into the list box?

The other category **Optimizing General Design** has not been published yet:

- Cluster Read: How do you provide high performance access to large chunks of data via an object/relational access layer?
- Bundled Write: How do you speed up the process of writing dirty objects to the database?
- Store for Forward: If you have too much data to wait for the transfer to a remote database, how do you shorten waiting time?
- Flat File Write: How do you write a large volume of data when you cannot wait for the database insertion?

Patterns

Pattern: Cluster Read

Example

You are programming a task that needs a large volume of data at a time. You know the structure of these data the moment you enter the use case in which you process them. Have a look at the invoice example below that is explained in more detail in the Accessing Relational Database Pattern Language[Kel+98a]. Now consider you want to build an high speed online browser for large invoices.

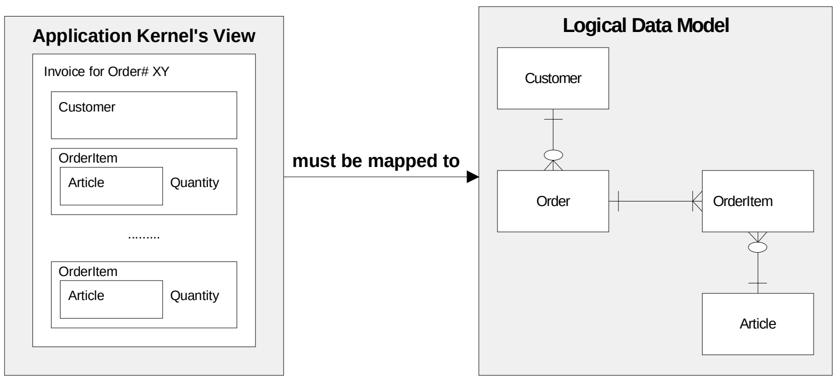


Figure 15: Part of an Order Processing System

It's not good idea to read an invoice object, dereference a customer proxy, dereference n order position proxies plus n proxies for each product. This would require 1 + 1 + n + n calls to the database over the network, consuming from 200 to 500 Milliseconds each.

Problem

How do you provide high performance access to large chunks of data via an object/relational access layer?

Forces

Performance versus complexity and cost: relational databases are missing a concept of clusters across multiple records that allows reading larger chunks of data across tables at a time - at least they do not support it at the level of query languages like SQL. Building something that is able to handle larger chunks of date or clusters will increase the complexity of your access layer.

Solution

Write a stored procedure or an access layer module that contains a series of SQL queries that get exactly the data that you want - all at the same time.

Structure

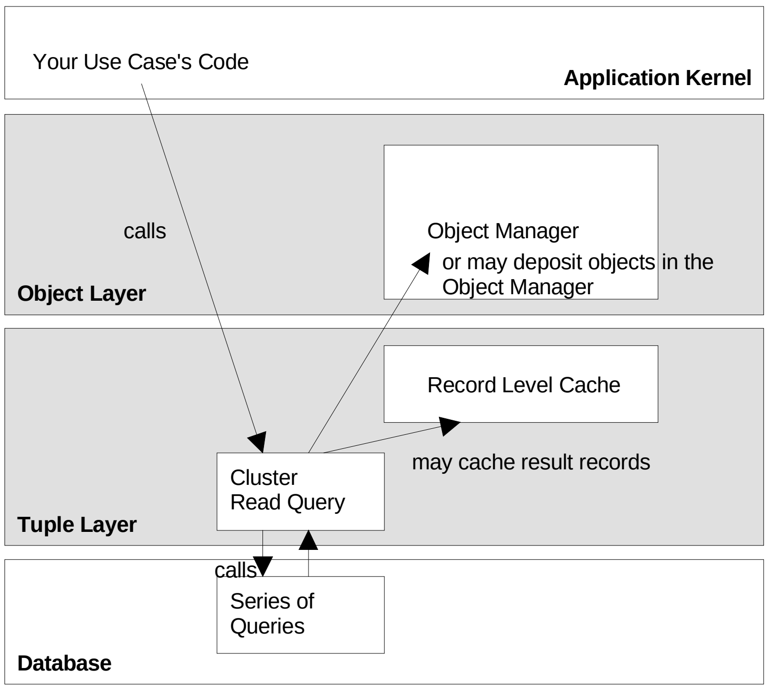


Figure 16: Calling a Cluster Read Query

You call the cluster read operation (usually a module of the tuple layer) directly from the application kernel. The module will deposits its results in a record level cache below the object manager. It might as well create objects from the results and place them directly in the object manager depending on the complexity of your mapping.

Example Resolved

Applying the Cluster Read pattern to the above example will yield only one database request plus a reduced number of database accesses, depending on the physical structure of the database.

Consequences

Performance: You economize on database calls, potentially over a network and get rid of lots of call overhead. The pattern can speed up complex use cases by up to 90%.

Orthogonality of the persistent language interface: Using this pattern introduces a new kind of call to the application kernel's interface, that is a direct call to a cluster read. This somewhat makes persistence less orthogonal - which is not so nice from the perspective of interface esthetics.

Maintenance: If you "hack" cluster reads based directly on the physical database scheme, you will get a maintenance problem when the physical structure of the database changes. This is affordable as you usually only need a few dozen cluster reads even in large scale applications.

Related Patterns

Cluster Read is a form of request bundling and so resembles Bundled Write. It uses exactly the idea behind Logical Views, so it is pretty common in all host based transaction systems that handle large amounts of data for single use cases. Cluster read may also be used with optimization patterns like denormalization, overflow tables and so on.

Known Uses

Reading data by clusters and request bundling are ubiquitous. The basic idea of Clustering is used in many storage subsystems. The pattern in this form is used in the Phoenix Persistence subsystem [Sta+97] by EA Generali. Complex stored procedures are used for similar reasons.

Pattern: Bundled Write

Example

You write a long transaction at user code level. Once you run your code, you load objects from the database into your object manager's cache. You manipulate the objects, and then you have say 55 dirty objects in your object manager [You+95]. If you start a naive traversal of the object manager telling each object to *"write itself down to the database"* this will result in at least 55 calls to the database with all the call overheads discussed in the Cluster Read pattern. Happy waiting!

Problem

How do you speed up the process of writing dirty objects to the database?

Forces

Performance: Implementing some bundling here is absolutely necessary - it's not even a matter of discussing this against implementation cost. If you don't do it, performance will be below anything that's reasonable.

Solution

Pick up all the statements generated by query objects (physical views) and send them to the database as a single packet of statements

Structure

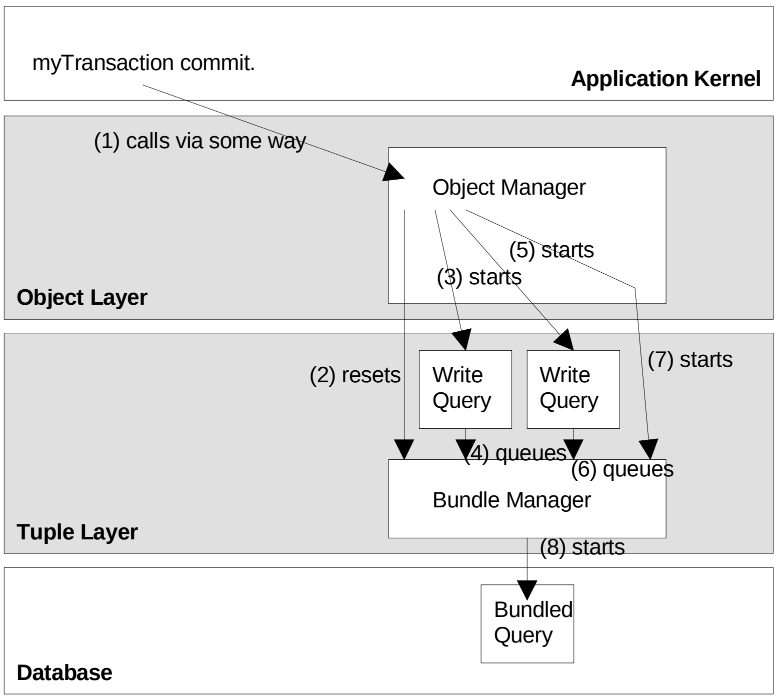


Figure 17: Calling a Cluster Read Query

The tuple layer needs to support bundling write requests. This bundle manager needs reset, queueStatement, start, and getErrorState operations. Result handling needn't be complicated as you never expect a result for an update or insert statement except an error code.

Example Resolved

Using the Bundled Write pattern will result in a single bundled statement issued to the database resulting in improved performance.

Consequences

Performance: will be reasonable. By the way - what do you call a pattern that MUST be applied in a client/server environment?

Cost: The bundle manager is straightforward and adds only little code.

Related Patterns

Cluster Read is a form of bundling request. The interesting questions is, why can't you use the identical implementation for both Cluster Read and Bundled Write. The answer is: If you request a result of a read you want it immediately and not at some later time, when the access layer decides to execute your query and get your objects. When you flush the object manager at the end of a transaction (i.e. when Bundled Write occurs), you have to wait the dirty objects must be written in a single logical transaction.

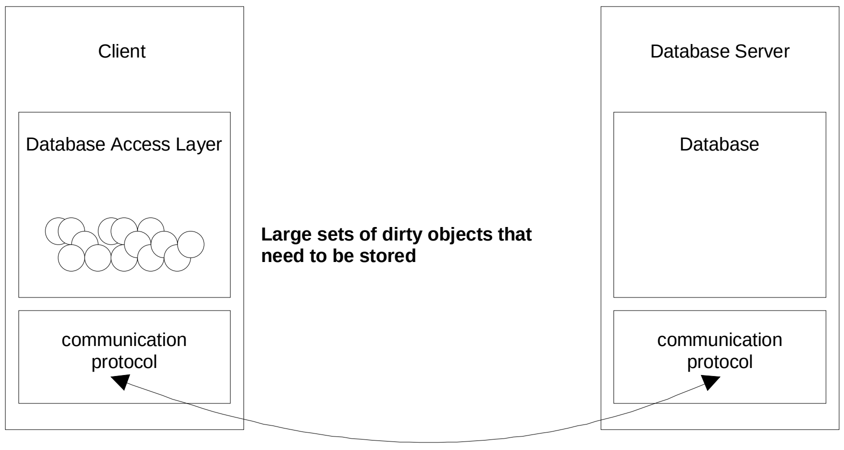
Known Uses

Most object/relational access layers use this pattern, e.g. TopLink [TOP97a] or the HYPO Project [Kel+98b]

Pattern: Store for Forward

Example

Imagine you are using an object/relational access layer and you are committing a transaction that contains some 100 or more changed objects that need to be written to the central remote database. The updates will take say 20 seconds, even if you use bundled write, and you don't want to keep your user waiting for such a long time.



Problem

How do you prevent long waiting times when your user has changed many objects

Forces

Performance: Even if each of the update statements is processed with near optimal performance by the database, no user likes to wait for 3 or more seconds. You have to come up with a way to improve the performance that is felt by your user.

Correctness: On the other hand it might be necessary that all objects of the transaction you are committing are written into the central database before you can start a new transaction. In this case it seems your user has to wait.

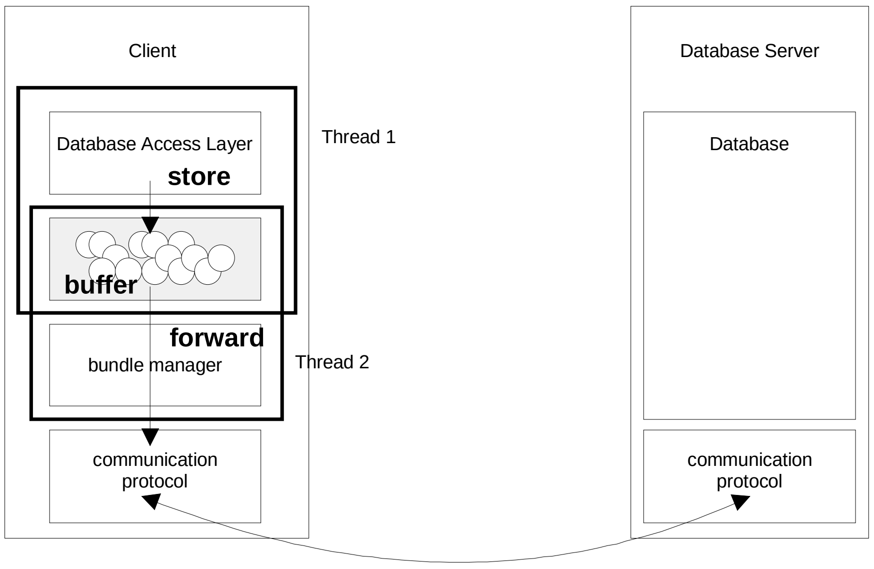
Security: Keeping data on a well administered server is safer than keeping them on a personal computer - even for minutes. On the other hand, the probability that a PC crashes is not much higher than the probability that somebody misspells something on a paper form and has to write that paper form again by hand, but. users are less tolerant with computers.

Cost and complexity: Whatever solution you are planning - it should be affordable and simple.

Solution

Store your data in a local buffer and give control back to your user. Have a separate thread of execution forward your data from the buffer while your user works on the next task.

Structure



The user process (Thread 1) stores the data to be written in a local buffer. A background process (Thread 2, for example a bundle manager) forwards them to the remote database.

Consequences

Performance and user acceptance of a system: If you store your data locally, the speed you gain from this can come close to the I/O processing rate of the computer you are working on. This is more than fast enough for most business systems.

Correctness: You have to make sure, the next transaction your user is working on does not collide with data that you have stored for forward. The object sets should not contain any common objects unless you flush your client cache and risk a time stamp collision.

Security: Keeping data on a PC for a few seconds is save enough in most cases - even if host acolytes will give you a bad time for it.

Cost and Complexity: Most store for forward schemes can be made so simple that cost is affordable. If you use bundled write for example, your bundle manager can work in a separate thread, synchronized with your application. This is not much effort.

Variants

Replicated Databases must use a store for forward scheme in order to work properly. You can implement your own store for forward schemes by using *synchronized flat files*. You can also store your data to a *local database* and use a forwarding job in a second thread to do the forwarding for you.

Related Patterns

The pattern can be combined with bundled write.

Known Uses

The pattern has been used extensively in a fat client banking application by Genesys [Sta98]. Phoenix uses it for the forwarding of Error Protocols, in case the database connection is broken.

Pattern: Flat File Write

Example

Imagine you write a batch job that processes life insurance policies. You have to process about 60.000 policies in one or two night batches. This means you have something like 700 milliseconds per policy. An analysis of the number of tables you have to update indicates, that each policy will cause about 70 inserts into a relational database - at 100ms per update this is 7 seconds for the updates alone - even on a very powerful host computer.

Problem

How do you handle output to a database when your database seems far too slow?

Forces

Performance of relational database against advantages of using them: There are situations when a relational database system simply seems too slow at a first glance, but you don't want to go back to using hierarchical database systems for the whole system just to support that one batch job. Loading a relational database from a flat file is usually at least one order of magnitude faster than individual inserts and updates.

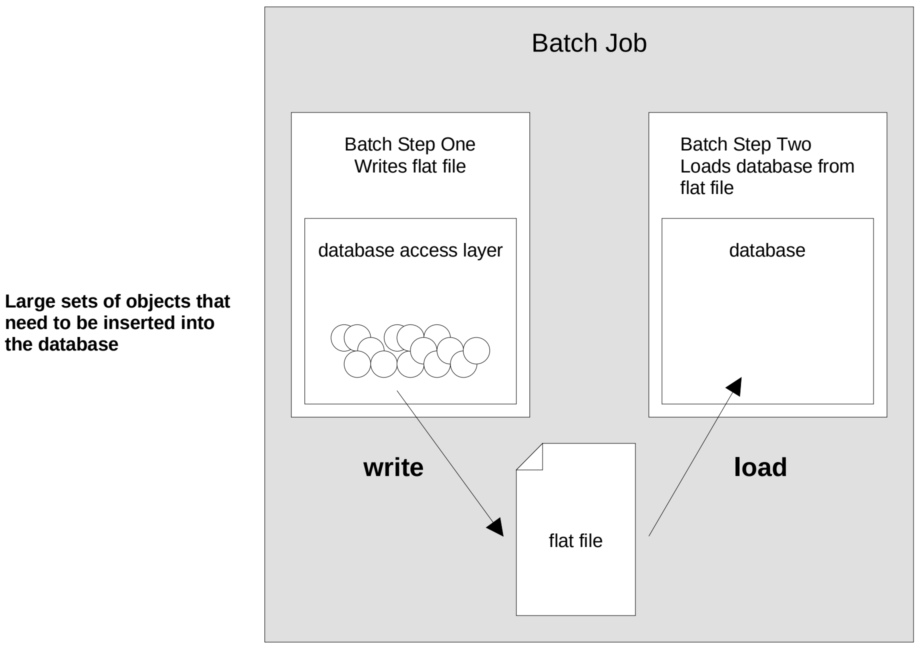
Correctness versus Performance: Your results have to be correct - you cannot afford to lock large regions of the database for days and you cannot afford to allow errors.

Cost and Complexity: The solution should work with normal hardware at reasonable cost.

Solution

Write the records to a transaction secured flat file (VSAM or the like) and load the database from that file later. Note that this solution works for inserts only. In case of updates you have to merge your file with another file that contains the unloaded content of your database.

Structure


Batch Job
Batch Step One Writes flat file
database access layer
Batch Step Two Loads database from flat file
database
Large sets of objects that need to be inserted into the database
write
load
flat file

Consequences

Performance: Your performance problem will be solved in most cases.

Correctness: Your batch should refer to only the data that you are processing, otherwise you will have pending updates in your flat files that will result in lost updates.

Cost and Complexity: You have to design a database access layer that is able to redirect its output to flat files. This is straightforward and not too expensive to implement.

Related Patterns

You can see this pattern as a specialized version of store for forward. You use a very fast way to store your data in a form that is not the final one and forward them to the final destination (the relational database) later.

Known Uses

Many large scale batch jobs use the pattern. We will use it in the Phoenix project for the batch job described in the example. Many other insurance projects have used it so far.

Building the Access Layer

Another set of existing patterns for object/relational access layers deals with questions of the software development process. They provide solutions for questions like "how do you do X" or "when is it best to do X".

Local Roadmap

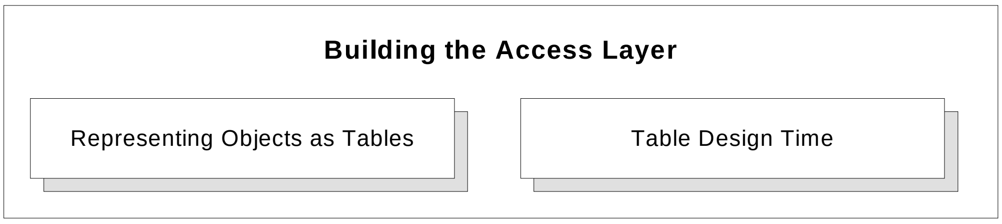


Figure 18: Local Roadmap: Building the Access Layer

Pattern List

Some patterns that can be found in literature are:

- Table Design Time [Bro+96]: When is it best to design a relational database during OO development.
- Representing Objects as Tables [Bro+96]: How do you map an object structure into a relational database schema.

Seeing the trouble that large projects have with a central database group that owns the object/relational mapping process, there must be more process patterns.

Problem/Solution Index of the Language

Architecting an Object/Relational Access Layer

Problem	Solution	Pattern Name	Source
What is a good overall architecture for Business Systems?	Build a layered architecture consisting of three layers: A user interface layer, a domain object layer plus a persistence layer.	Layered Architecture for Business Systems	[Ren+97]
What is a good structure for a persistence subsystem?	Build your system consisting of two subsystems that form a layered structure. The upper layer, called the object layer, encapsulates the concepts of object-orientation while the lower layer called the storage manager offers a high level interface on top of your physical storage devices or file system.	Two Layer Persistency Subsystem	This paper
How do you provide an easy to use interface to your physical database tables?	Encapsulate every table and every view with a wrapper class. Use these classes to encapsulate Overflow Tables and other database optimization techniques. To provide a uniform interface derive the wrapper classes from a protocol class.	Physical Views	[Kel+98a]
How do you link a database access layer to a transaction based host database server?	Use a communication agent between your client and host computer that does request bundling. See page 10.	Host Access	This Paper

Designing the Object Layer

Problem	Solution	Pattern Name	Source
How do you represent an object's individuality in a relational database?	Assign an the objects a synthetic key that accompanies the object from birth to destruction. Bury the key with the object.	Object Identifier	[Bro+96], [Atk+89]
How do you prevent all related objects being loaded whenever you touch one object that has relations to many others?	Use a Smart Pointer (or Proxy) containing an Object Identifier plus a memory pointer that is instantiated with NULL whenever a Proxy is instantiated.	Proxy	[GOF95], [Bro+96], [Mey96].
How do you preserve object identity?	Create a cache of objects per database client process. Base the cache on a container that maps Object Identifiers to pointers to Objects (Proxies).	Object Manager	[You+95, pages 291-292, Max96], [Bro+96]
How do you handle transactions at a user code level?	Make a transaction an object. Give it operations like begin, commit, rollback.	Transaction Object	[Kel+97]

Accessing Relational Databases

Problem	Solution	Pattern Name	Source
How do you provide an easy to use interface to your physical database tables?	Encapsulate every table and every view with a wrapper class. Use these classes to encapsulate Overflow Tables and other database optimization techniques. To provide a uniform interface derive the wrapper classes from a protocol class.	Physical Views also known as Query	[Kel+98a] [Bra+96]

Moving Attributes to and from the Tuple Layer

Problem	Solution	Pattern Name	Source
How do you move object's attributes between layers of a system?	Extend a single class over both layers, forming a *Multilayer Class*. Unambiguously assign every member of the class to one layer, using a naming convention. Relieve the Multilayer Class from as many lower layer responsibilities as possible and encapsulate them in separate classes. Prefer to use call dependencies over inheritance.	Multilayer Class	[Col+96b]
How do you move objects' attributes between layers of a system?	Create another object for each upper layer object that has the responsibility to move the attributes up and down.	Class Broker	[Bro+96]
How do you move object's attributes between layers of a system?	Create one object for all upper layer objects that has the responsibility to move the attributes up and down.	Central Broker	[TOP97a,TOP 97b]

Mapping Objects to Tables

Problem	Solution	Pattern Name	Source
How do you map aggregation to relational tables?	Put the aggregated objects' attributes into the same table as the aggregating object's	Single Table Aggregation	[Kel97]
How do you map aggregation to relational tables?	Use a separate table for the aggregated object. Insert an Object Identifier 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	Foreign Key Aggregation	[Kel97]

Problem	Solution	Pattern Name	Source
How do you map an inheritance hierarchy of classes to database tables?	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.	One Inheritance Tree One Table	[Kel97]
How do you map an inheritance hierarchy of classes to database tables?	Map the attributes of each class to a separate table. Insert an Object Identifier into each table to link derived classes rows with their parent table's corresponding rows.	One Class One Table	[Kel97]
How do you map an inheritance hierarchy of classes to database tables?	Map the attributes of each class to a separate table. Add the attributes of all classes the class inherits from to a class's table.	One Inheritance Path One Table	[Kel97]
How do you map an 1:n association to relational tables?	Insert the owner object's OID into the dependent objects table. The OID may be represented by a database key or a Object Identifier.	Foreign Key Association	[Kel97]
How do you map n:m associations to relational tables?	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 [Kel97].	Association Table	[Kel97]
How do you map objects to a relational database?	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.	Objects in BLOBs	[Kel97]

Optimizing Performance

Problem	Solution	Pattern Name	Source
How can you manage to read and write object clusters with a single page database access when you have a parent/child relation?	Fill up a parent entities database page with child entities records until you reach the next physical page limit.	Denormalization	[Kel+97]
You have followed the Denormalization pattern's advice and have denormalized a relation. What do you do with those objects that have more dependent objects than the number that you did integrate into the parent object's table?	Use a second table, a overflow table, that contains another physical database page full of child entities records.	Overflow Table	[Kel+97]
How can you manage to read object clusters with a single page database access when you need to read data from a parent object's table?	Replicate those parts of the parent entity in the child entity that you need for a use case. Replicate only stable data that are not subject to frequent updates.	Controlled Redundancy	[Kel+97]
What kind of database or object level views should you use for filling list boxes?	. Views for list boxes should contain the data needed in the list box and the primary key to access the object that you intend to select from the list box.	Narrow Views	[Kel+97]
How do you speed up filling of list boxes and how do you prevent unnecessary data from being loaded into the list box?	Load data in chunks that allow a reasonable response time. A rule of thumb is 30-50 records for a C/S system. This is equivalent twice the number of lines in a list box.	Short Views	[Kel+97]
How do you provide high performance access to large chunks of data via an object/relational access layer?	Write a stored procedure or an access layer module that contains a series of SQL queries that get exactly the data that you want – all at a time.	Cluster Read	This paper

Problem	**Solution**	**Pattern Name**	**Source**
How do you speed up the process of writing dirty objects to the database?	Pick up all the statements generated by query objects (physical views) and send them to the database as a single packet of statements	Bundled Write	This paper
If you have to much data, to wait for the transfer to a remote database. How do you shorten waiting time?	Store your data in a local buffer and give control back to your user. Have a separate thread of execution forward your data from the buffer while your user works on the next task.	Store for Forward	This paper
If you have to write so many data, that you cannot wait for the database inserts, what do you do?	Write the records to a transaction secured flat file (VSAM or the like) and load the database from that file later.	Flat File Write	This paper

Building the Access Layer

Problem	**Solution**	**Pattern Name**	**Source**
How do you map an object structure into a relational database schema	Begin by creating a table for each persistent user-defined object in your object model ... for the rest see [Bro+96]	Representing Objects as Tables	[Bro+96]
When is it best to design a relational database during OO development.	Design the tables based on your object model after you have implemented it in an architectural prototype but before the application is in full-stage production	Table Design Time	[Bro+96]

Glossary

The following is a glossary of terms that might not be familiar to people who do shiny new stuff only and have never been confronted with the “old world” of host systems.

CICS:	CICS is a general-purpose online transaction processing (OLTP) software system by IBM. CICS is an application server that runs on a range of operating systems from small desktops to large mainframes, and which meets transaction-processing needs, whether you have thousands of terminals or a client/server environment with workstations and LANs. CICS, as a transaction system takes care of the security and integrity of your data while looking after resource scheduling, thus making effective use of your resources. CICS integrates basic software services required by OLTP (Online Transaction Processing) applications. For more details see: http://www.software.ibm.com/ts/cics/
Host:	In IBM and perhaps other mainframe computer environments, a host is a mainframe computer (which is now usually referred to as a "large server"). In this context, the mainframe has intelligent or "dumb" workstations attached to it that use it as a host provider of services. (This does not mean that the host only has "servers" and the workstations only have "clients." The server/client relationship is a programming model independent of this contextual usage of "host.") (Source: http://whatis.com/)
IMS:	IMS is a family of products by IBM. IMS/DB is a hierarchical database system (see also [Dat95] for some more information). IMS/TM (TM for Transaction Monitor) is an online transaction processing system like CICS, just another product line. Many mainframe shops started as CICS or IMS shops. In the meantime many run both systems. For more details see: http://www.software.ibm.com/data/ims/
ODBC:	Open Database Connectivity (ODBC) is a standard or open application programming interface (API) for accessing a database. By using ODBC statements in a program, you can access files in a number of different databases, including Access, dBase, Excel, and Text. In addition to the ODBC software, a separate module or driver is needed for each database to be accessed. The main proponent and supplier of ODBC programming support is Microsoft. ODBC is based on and closely aligned with the X/Open standard Structured Query Language (SQL) Call-Level Interface. It allows programs to use SQL requests that will access databases without having to know the proprietary interfaces to the databases. ODBC handles the SQL request and converts it into a request the individual database system understands. (Source: http://whatis.com/)
Persistence:	Is the ability of objects to survive termination of the process they were created in. Or in other words the property of a programming language where created objects and variables continue to exist and retain their values between runs of the program
Recovery:	In the event of an application or system failure (for example, if there is a power loss and the computer system shuts down), when the system restarts, any uncompleted work that was in progress at the time of shutdown, including changes to data, must be backed out to a point where the system was last in a consistent state. This is called Recovery.
Transaction Monitor:	A program that manages or oversees the sequence of events that are part of a transaction is sometimes called a transaction monitor. When a transaction completes successfully, database changes are said to be committed; when a transaction does not complete, changes are rolled back. In IBM's CICS product, a transaction is used to mean the instance of a program that serves a particular transaction request. (Source: http://whatis.com/transac.htm)

VSAM:	Virtual Sequential Access Method is a file management system for IBM's larger operating systems, including its primary mainframe operating system, MVS (Multiple Virtual Storage), now called OS/390. Using VSAM, an enterprise can create and access records in a file in the sequential order that they were entered. It can also save and access each record with a key (for example, the name of an employee). Many corporations that developed programs for IBM's mainframes still run programs that access VSAM files (also called data sets). VSAM succeeded earlier IBM file access methods, SAM (Sequential Access Method) and ISAM (Indexed Sequential Access Method). Today, although VSAM is still provided in support of legacy applications, IBM emphasizes DB2, a relational database product, and many customers use database products from Oracle, Sybase, Computer Sciences, and other companies. (Source: http://whatis.com/)

References

[Atk+89] **Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, Stanley B. Zdonik:** *The Object-Oriented Database System Manifesto*. in *"Deductive and Object-Oriented Databases"*, Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD'89), pp. 223-240

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

[Bra+96] **John Brant, Joseph Yoder:** *Reports*, in "Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences„Washington University, Department of Computer Science, Technical Report WUCS 97-07, February 1997.

[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.

[Bus+96] **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal:** *Pattern Oriented Software Architecture - A System of Patterns*, Wiley 1996.

[Col96a] **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/

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

[Col+96b] **Jens Coldewey, Wolfgang Keller:** *Multilayer Class*, in „Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences„ Washington University, Department of Computer Science, Technical Report WUCS 97-07, February 1997.

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

[GOF95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**: *Design Patterns, Elements of Reusable Object-oriented Software*, 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

[Hei98] **Peter M. Heinckiens**: *Building Scalable Database Applications*, Addison-Wesley 1998.

[Kel97] **Wolfgang Keller**: *Mapping Objects to Tables: A Pattern Language*, in „Proceedings of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany, Siemens Technical Report 120/SW1/FB 1997.

[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.

[Kel+98a] **Wolfgang Keller, Jens Coldewey**: *Accessing Relational Databases: A Pattern Language*, in Robert Martin, Dirk Riehle, Frank Buschmann (Eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998.

[Kel+98b] **Wolfgang Keller, Christian Mitterbauer, Klaus Wagner**: *Object-oriented Data Integration: Running Several Generations of Database Technology in Parallel*; in Akmal Chaudhri, Mary Loomis (Eds.): Object Databases in Practice, Prentice Hall 1998.

[Max96] **John Maxfield:** *A Distributed Virtual Environment for Synchronous Collaboration in Simultaneous Enginnering*, Technical Report, The Keyworth Institute of Manufacturing and Information Systems Engineering, 1996.

[Mey96] **Scott Meyers:** *More Effective C++*; Addison-Wesley 1996.

[Nob97] **James Noble:** *Basic Relationship Patterns*, in „Proceedings of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany, Siemens Technical Report 120/SW1/FB 1997.

[ODMG93] **Rick G. G. Cattell (Ed.) et. al.**: *Object Database Standard (ODMG 93);* Morgan Kaufmann Publishers, 1993.

[POE97] **Frank Thelen, Jörg Beckert:** *POET SQL Object Factory - Technical Overview*; http://www.poet.com/sql_tech_over.htm, POET GmbH, 1997

[Ren+97] **Klaus Renzel, Wolfgang Keller:** *Three Layer Architecture* **in Manfred Broy, Ernst Denert, Klaus Renzel, Monika Schmidt (Eds.***)* *Software Architectures and Design Patterns in Business Applications*, Technical Report TUM-I9746, Technische Universität München, 1997.

[Via+97] **Mauricio J. Vianna e Silva, Sergio Carvalho, John Kapson***: Patterns for Layered Object-Oriented Applications*, in „Proceedings of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany, Siemens Technical Report 120/SW1/FB 1997.

[Sta+97] **Herbert Staudacher, Michael Harranth:** *Dokumentation der Datenpersistierung von Phoenix*, Internal Technical Documentation, EA Generali, Vienna 1997.

[Sta98] **Herbert Staudacher:** *Personal Communications over many many coffees*, 1997-1998.

[TOP97a] **The Object People Inc.:** TOPLink Version 4.0 - A White Paper; http://www.objectpeople.com/, 1997.

[TOP97b] **The Object People Inc.:** TOPLink Version 4.0 - User Manual, 1997.

[Wal+95] **Kim Walden, Jean-Marc Nerson**: *Seamless Object-oriented Software Architecture*, Prentice Hall 1995.

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