

Relational Database Access Layers A Pattern Language

The Key Patterns

Wolfgang Keller, Jens Coldewey
sd&m – software design & management GmbH & Co. KG;
Projekt ARCUS;
Thomas-Dehler-Str. 27 – D 81737 München - Deutschland;
Email: {wk, jensc}@sdm.de

Abstract

The Relational Database Access Layer helps you design applications that use relational databases and also reflect the relational calculus at a business object level. Such applications are known as *data driven* or *representational* [Mar95]. The systems need not be object-oriented, you may also use a 3GL. Hence, the pattern language passes over mapping inheritance and polymorphism.

This paper comprises the framework pattern and the key implementation patterns. The PLoP proceedings [Kel+96b] contain the complete pattern language, including adapters and optimization patterns.

Introduction

Many business information systems have a simple data model. Though they may have thirty entities or more, you rarely find much inheritance or complex associations. Yet there are complex use cases, you usually encapsulate in an application kernel. It is a good idea, to use the relational calculus for modeling these systems.

Consider the excerpt of an order management system, shown in the lower part of Figure 1. It models the entities you need to process the invoice, shown in the upper left corner. The excerpt complies to the *Third Normal Form* (3NF); a level of factorization, data analysts often use [Dat94]. Suppose, you have used this logical data model to define your physical database tables. The system will work correctly, but you shall come across bad performance.

Profiling the system you detect many superfluous database operations. You also find slow database operations caused by large joins or by moving unnecessary large amounts of data. To increase performance you denormalize the physical data model. A statistical analysis of the database's contents yields that ninety percent of the Orders have no more than five OrderItems. Therefore you decide to store the first five OrderItems in the Order table. To cover the remaining ten percent you create an OrderItemOverflow table as depicted in the upper right of Figure 1. Furthermore you integrate the Article attributes ArtPrice and ArtName into the Order table. The resulting database design allows to read ninety percent of the invoices with two database accesses: one to the Order table and on to the Customer table. Besides you have eliminated all joins.

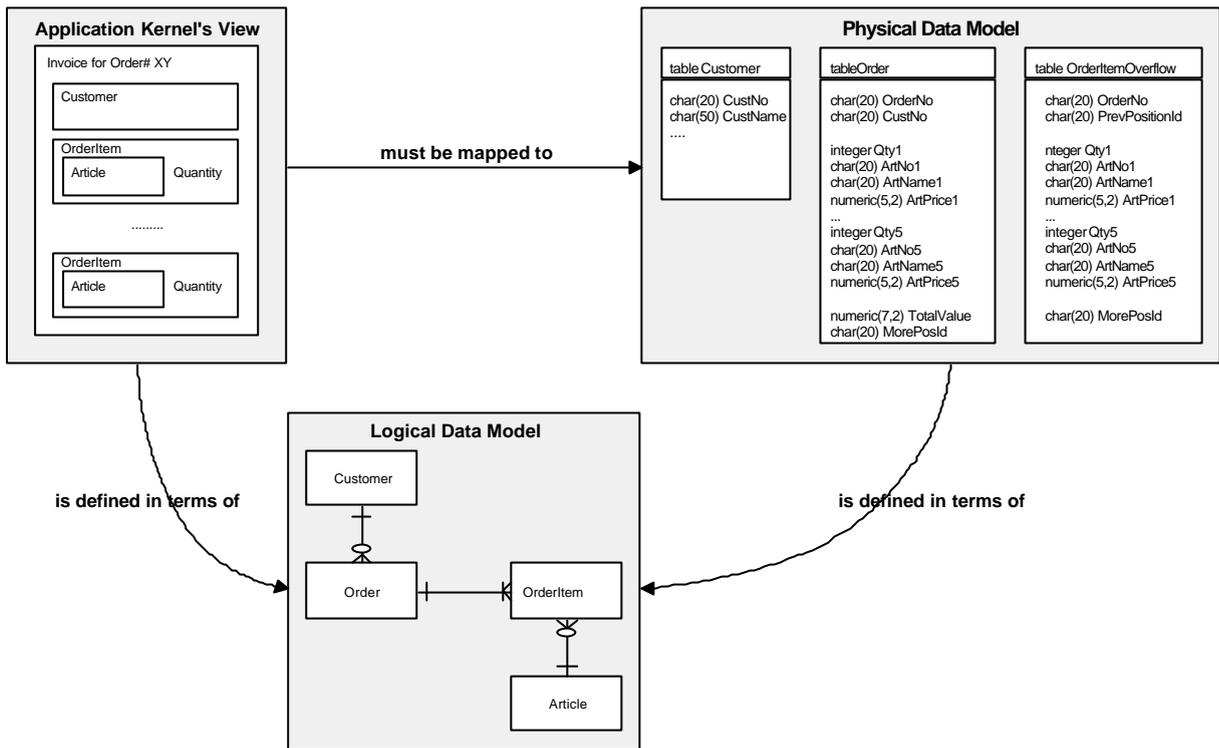


Figure 1: Part of an Order Processing System

Now assume, you have embedded SQL-statements within the application kernel code. To adapt the code to the new table schemes you have to rewrite large portions of it. Furthermore, handling overflow tables lets the SQL code explode. Worst of all, you have to repeat this procedure for every improvement of the database structure.

Pattern Language Map

The „Relational Database Access Layer“ framework pattern defines roles and responsibilities for its components. It also describes the three key abstractions Hierarchical Views, Physical Views and Query Broker.

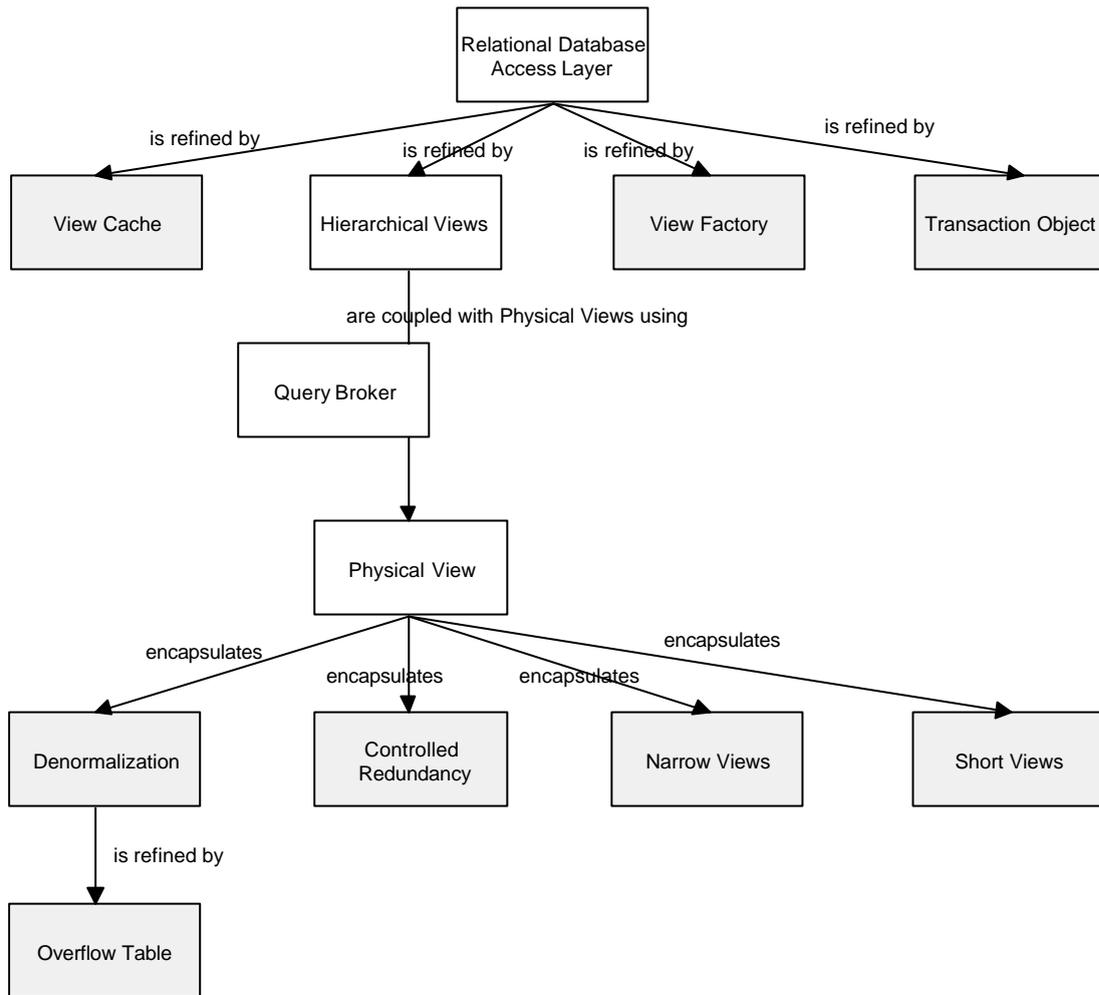


Figure 2: A Map of the pattern language. Unshaded boxes denote the key patterns presented in this paper. Shaded boxes are contained only in the PLoP Proceedings [Kel+96b]

Related Work on Database Access Layers

This framework provides a database view interface for applications that use their database in a relational fashion. Other persistence frameworks and pattern languages deal with full object persistence. These are *Object to Relational Access Layers* [Bro+96, Col+96, Kel+96a], using a relational database and *Object Access Layers* using an object database [Col97]. Figure 3 illustrates the relationship between the different access layers

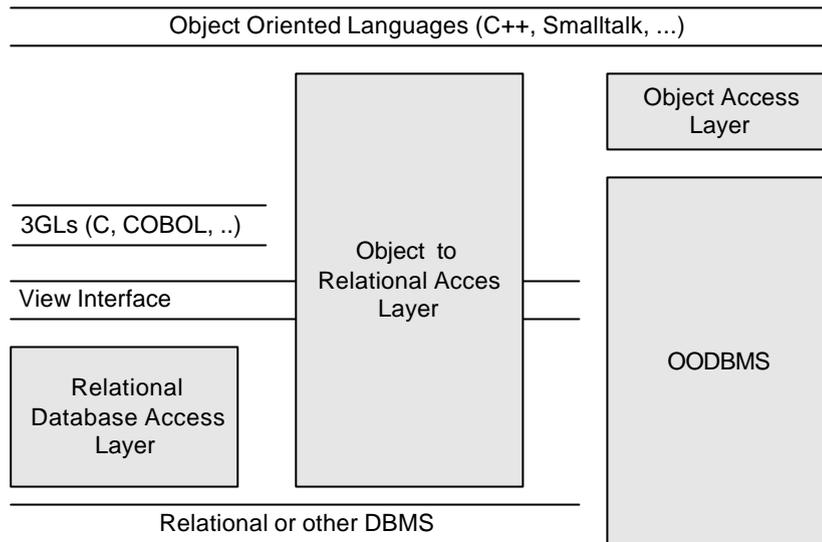


Figure 3: The three different kinds of database access layers

Notational Conventions

We use OMT [Rum+91] for object diagrams. Another Pattern references a related pattern. If a pattern reference is followed by a citation, such as [GOF95], you can find it in the cited paper.

The Framework Pattern

Pattern: Relational Database Access Layer

Context

You are writing a business information system like the preceding order processing system. The relational calculus is an appropriate representation of the domain logic. The resulting data model is simple and uses inheritance sparingly. The effort of mapping the relational model to an object-oriented representation is high compared to the gains.

Problem

How do you access the relational database?

Forces

?? *Separation of concerns versus cost:* Database programming is complex and so is application logic. Both call for clever solutions. Mixing them will add up to more than just the combined complexity. The easiest way is to separate application programming from database programming. Both parts will be easier to implement and to test. On the other hand the introduction of new layers increases the number of classes and raises design and implementation effort. The cost has to pay off with increased maintainability and easier performance tuning.

?? *Ease-of-use versus power:* If you decide to encapsulate the database, the resulting 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 encapsulation should be easy to use but still powerful enough for your project.

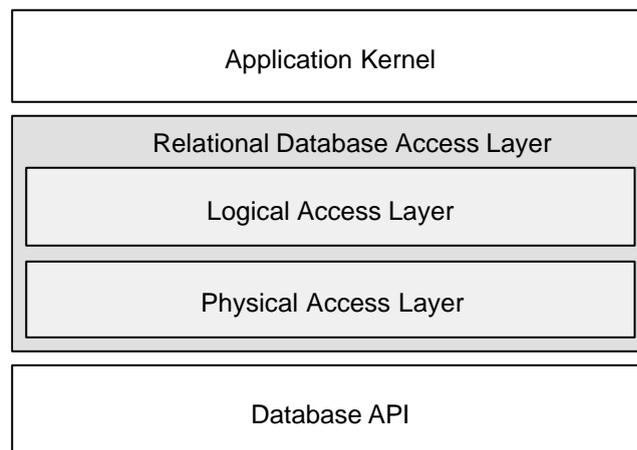
?? *Performance:* Database tuning is crucial to achieve acceptable performance of a business information system. Since a database is several orders of magnitude slower than a main processor, tuning actions will concentrate on database access. Tuning is an iterative process. To optimize database access you may change physical parameters of the storage, 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 the application kernel remains untouched. Still, the more flexible a system is, the more complex it will be.

?? *Legacy systems versus optimal design:* 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. However, the structure of legacy data rarely fits your needs - if there is a structure at all. 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.

Solution

Use a layered architecture consisting of two layers. The Logical Access Layer provides the stable application kernel's interface, while the Physical Access Layer accesses the database system. The latter may adapt to changing performance needs. Use a Query Broker to decouple both layers.



Structure

Figure 4 shows the classes of the Relational Database Access Layer. The Logical Access Layer provides classes for caching and transaction management. The Physical Access Layer represents the

interface to the database system. The latter splits into the physical views, representing data access, and the Database class, which encapsulates administrative calls. Hardwired logic or - even better - a Query Broker mediates between the logical and the physical access layer.

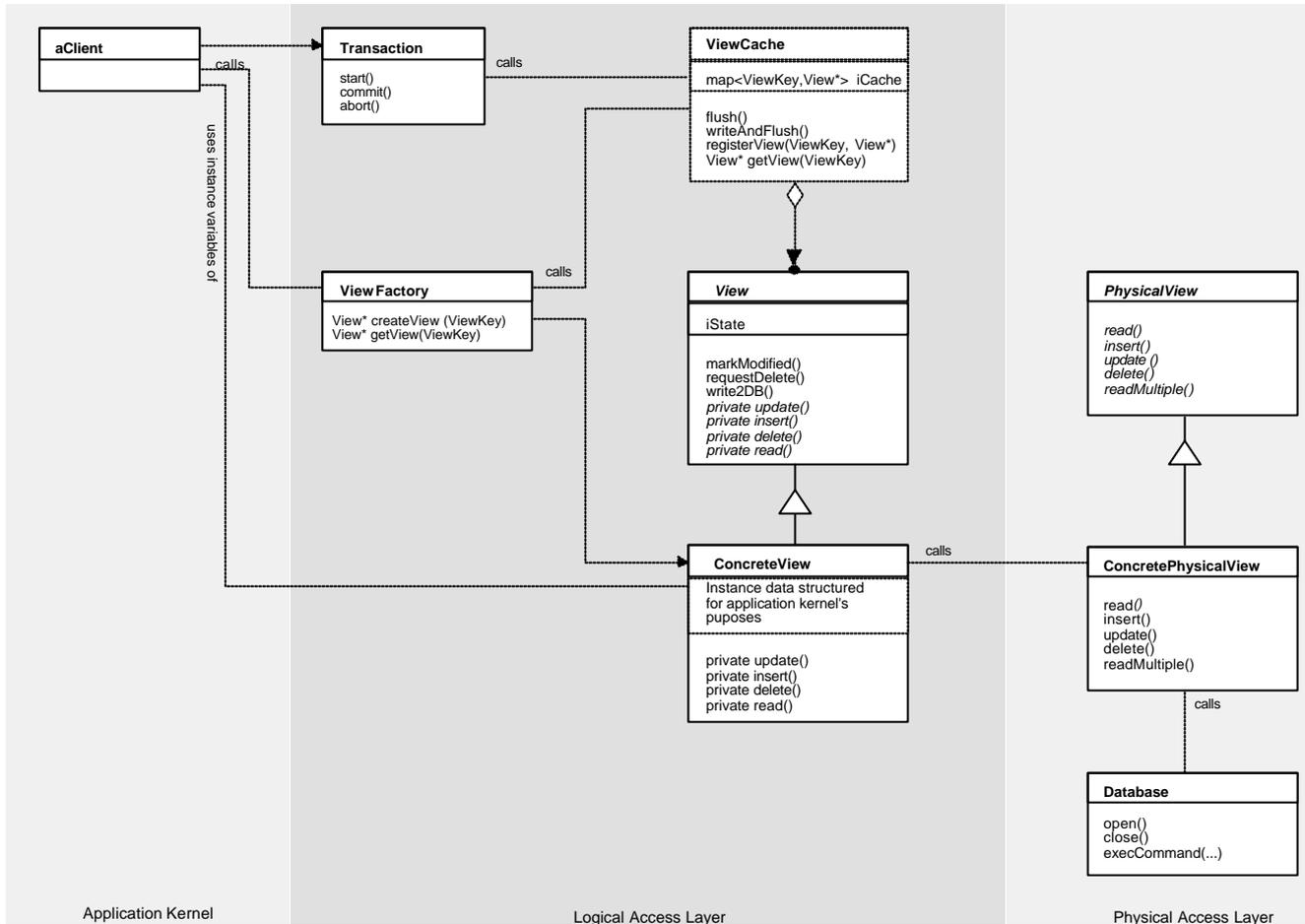


Figure 4: The structure of the Relational Database Access Layer framework. The Client accesses only classes of the Logical Access Layer, which, in turn, use the Physical Access Layer to connect to the database.

Participants

Transaction

?? offers an interface that allows start(), commit() and rollback() of transactions.

?? is created at the beginning of every transaction. It is destroyed after a commit() or an abort().

This usage resembles the transaction object defined in [ODMG96, chapter 2.8].

ViewFactory

?? *delivers data identified by a key*. Therefore it offers `createView()` and `getView()` methods to create new Views and activate existing ones. These two methods are the only way for the Client to get a reference to a View.

?? *uses a View Cache* [Kel+96b] to avoid creating Views if they already exist in the context of the current transaction.

?? *allows predicates to identify ConcreteViews*. An abstract key class `ViewKey` provides a standard interface for all keys.

?? *is a Singleton* [GOF95].

ViewCache

?? *prevents data from being loaded twice*. It is a keyed Container of Views forming the access layer's cache.

?? *offers the writeAndFlush method*, which writes all modified Views to the database using their `write2DB` method. The Transaction object calls `writeAndFlush` when it commits. Upon abort Transaction calls the `flush()` method to clear the ViewCache instead.

ConcreteView

?? *is a Hierarchical View on the logical data model*. There are several `ConcreteView` classes, each of them tailored for one or more use cases of the application kernel. Members of `ConcreteViews` are application data types, not raw database types¹.

?? *knows how to write itself to the database* using `ConcretePhysicalViews`. The `ConcreteView` keeps track of its internal state and calls private `update()`, `insert()` or `delete()` methods if it receives a `write2DB()` message from the `ViewCache`. You may use a hard coded calls to suitable `ConcretePhysicalViews` or a Query Broker.

View

?? *defines the abstract protocol* for `ConcreteViews`. See also the Hierarchical View pattern.

?? *offers a markModified() method* to cause a database update when the current transaction commits.

?? *provides the requestDelete() method* to generate a database delete at the end of a transaction. Do not confuse this method with a destructor. While the destructor instantly eliminates the objects from memory, `requestDelete()` erases the object from the database (and from memory) at the

¹ Application data types are often used instead of raw database data types at application kernel level. They have additional methods to check their contents, reformat it, or format it for output. [Den91, chapter 5.2]

end of the transaction. To avoid dangling references, the `requestDelete()` method should be the only way to delete database records.

PhysicalView

?? *defines a uniform protocol* for `ConcretePhysicalViews`.

?? *bundles database access functions*, encapsulates database behavior and translates database error codes into application level errors².

ConcretePhysicalView

?? *wraps one physical database table*. It may also wrap database views. If the database does not support direct update of views, the `ConcretePhysicalView` also issues the appropriate write commands.

?? *also wraps database optimization* if you use Denormalization, Controlled Redundancy, or Overflow Tables. In this case a `ConcretePhysicalPhysical` view may map more than one table.

?? *may be generated* from meta information, such as the table structure of the database.

Database

?? *encapsulates the database management system*. Provides methods for starting database connections, issuing database commands and receiving results.

Dynamic Behavior

We shall discuss dynamic behavior with the patterns that implement the different aspects of the framework.

Implementation

?? *Treatment of Mass Updates*: Mass updates are statements of the form “update .. where”, which manipulate a set of records with a single query. It is hard to integrate these statements with the View Cache. Incorporating mass updates means: Perform a mass read into the `ViewCache`, manipulate single records, and write them back into the database one at a time. This solution is much slower than directly performing the task on the database.

?? *Batches* need special treatment. There is a set of patterns dealing with batch database access, which are waiting to be mined.

?? *Multiple Read Queries*: We have skipped multiple read queries. You can find further information in the Short Views and Narrow Views patterns.

² Most database errors are not meaningful on application level. Therefore it is a good idea to translate them. Additionally you may have to translate the error mechanism when the database uses return codes to signal errors and your application uses exceptions.

- ?? *Cursor Stability*: There is the theoretical possibility to submit mass read operations to a BFIM (before images) consistency check. This would provide level 2 transaction consistency (*cursor stability* [Gra+93]) instead of level 1 (*browse consistency*). Mass read operations are typically used to fill list boxes (see Short Views). They have the form „select <fields> from ... where“. Checking them for consistency at commit time would mean rereading all read records read during the transaction and comparing them to their before images. If only a single record differs you have to abort the transaction. This is not only a serious threat to performance - it also does not add any value to consistency. In most cases records used to fill list boxes do not play any role that could flaw the consistency of a task. Hence, it is usually sufficient to use browse consistency for data not involved in computations during the transaction.
- ?? *ConcretePhysicalViews and dynamic SQL*: If the database system supports dynamic SQL without runtime penalty, you may skip the *ConcretePhysicalViews* and use the Query Broker to generate the appropriate SQL statements. For static SQL the *ConcretePhysicalViews* provide the queries.
- ?? *Database Connections* should remain established as long as possible. Establishing a new connecting for every Transaction shall result in bad performance.
- ?? The use of database triggers and stored procedures containing business logic is strongly discouraged with this architecture. A View Cache will not be notified about autonomous changes in the database. Hence stored procedures may cause cache consistency problems. Similar problems arise with triggers: Since they work on the physical data model it is hard to transform them to the logical level of the application kernel. However, you may use restricted stored procedures to speed up data access (see Physical Views).

Consequences

- ?? *Separation of concerns*: The access layer forms a well-encapsulated subsystem for transactions, database access and caching. The application kernel uses a logical interface and needs no knowledge about database access.
- ?? *Effort*: Implementing a Relational Database Access Layer requires from 0,5 to 35 person years, depending on the features you include. Using generators and hard coded dependencies is cheaper than building maintenance tools and a Query Broker. Consider expected changes, time to market and the lifetime of your software before you decide for a variant.
- ?? *Ease-of-use*: The access layer does not transform the relational model to an object-oriented view. Therefore the application kernel has to cope with the relational view. You should carefully consider whether this *data driven approach* matches the application logic or not. Check whether your project does better with an Object to Relational Access Layer. It is not a good idea to save the effort for a more complex access layer when the application kernel does the required mapping implicitly .
- ?? *Flexibility*: If you use a Query Broker, you may maintain and tune the database by adding new *ConcretePhysicalViews* instead of modifying application kernel code. The application code remains stable while the underlying physical database changes for tuning.

?? *Complexity*: The access layer contains mostly simple classes. The Query Broker is the most expensive item as it contains a complex tree matching algorithm. Omitting it results in a simple layer of adapters but is less flexible.

?? *Performance*: You pay a minor run time penalty for the mapping. Anyway the access layer eases tuning and optimizes using caching and eases tuning. You spend on fast processor cycles and economize on slow I/O.

?? *Legacy data*: You may use the access layer to decouple the physical and the logical data models of existing applications. This is useful to reengineer legacy applications. First you insert a database access layer into the code, which is a single step with manageable risks. Then you start to rewrite the database and the application kernel in different projects.

Variants

?? *Omitting the ViewCache*: If you do not need long transactions, you may omit the ViewCache. This is a feasible approach for simple dialog systems supporting only the manipulation of a single record per transaction. However, you should use the ViewCache if the application kernel has a notion of transactions affecting more than one record.

?? A cache is the natural choice to implement user transactions on top of a transaction monitor, such as IMS, CICS, or UTM. Transaction monitors start a new transaction for every step the dialog takes while user transactions contain typically several dialog steps to complete. Using the ViewCache enables you to collect all write activities to the database that occur during a user transaction. They are later executed in a single technical host transaction preserving transaction integrity over multiple dialog steps of a transaction system.

?? *Using non-relational Databases*: The Physical Access Layer may also encapsulate non-relational databases and file formats, such as IMS-DB, CODASYL, or VSAM. You may even adapt to several different database technologies, thus hiding access to legacy data.

Known Uses

The VAA Data Manager specification uses this pattern together with editors for meta data and complex mappings for hierarchical database systems [VAA95]. The VAA Data Manger is derived from the Data Manager Architecture of Württembergische Versicherung [Würt96].

Denert sketches some basic ideas of the pattern language in [Den91, pp. 230-239]. Many projects at sd&m used the patterns in various variants including Thyssen, Deutsche Bahn, and HYPO Bank [Kel+96a].

The CORBA Persistent Object Services (POS) [Ses96] specify persistent objects that use a Broker (Persistent Object Manager) to write their data to arbitrary data stores (Persistent Data Services).

Related Patterns

The pattern is an application of Layers [Bus+96, pp. 31]. The View Factory is an application of the Abstract Constructor [Lan96]

[Bro+96] and [Col+96] describe how to extend the pattern to offer an object-oriented view of a relational database to the application kernel. Brown and Whitenack [Bro+96] use a broker to decouple the layers while [Col+96] describes a hard-wired approach..

Some Implementation Patterns

Pattern: Hierarchical Views

Example

Consider the detail of our order processing system shown in Figure 5. There may be use cases working with invoices having the structure depicted on the right side. Note that this invoice has a hierarchical structure with two levels of indirection. The use case may start with an order number and then navigate to the various items and their articles.

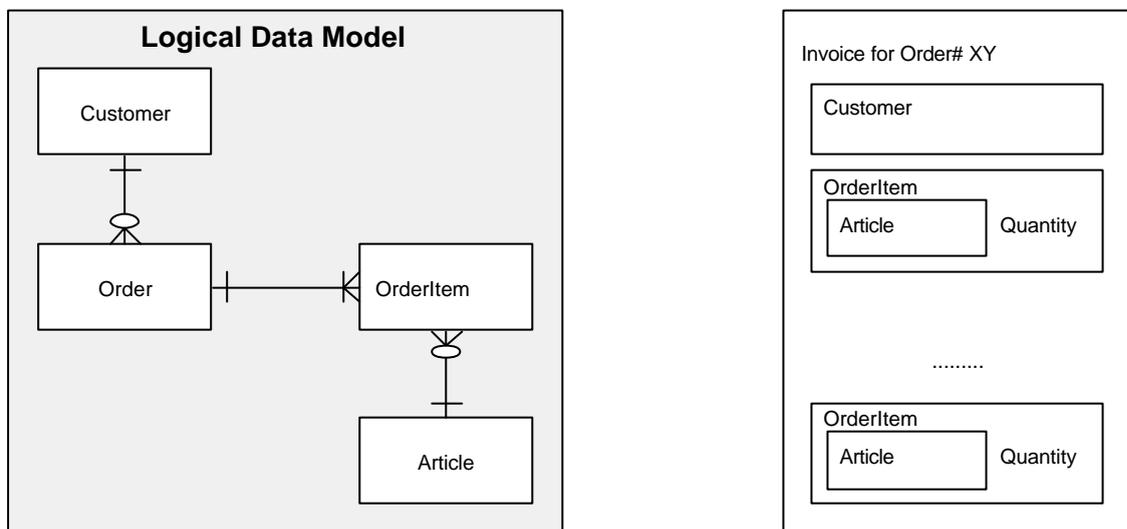


Figure 5: A detail of our order processing system's logical data model. The left side shows the E/R diagram in third normal form, the right side the structure of an invoice, composed of these entities.

Context

You have decided to use the Relational Database Access Layer to decouple your physical database from the logical data model of the application kernel.

Problem

What interface presents the database access layer to the application kernel?

Forces

?? *Complexity versus ease-of-use and development cost:* The interface should be easy to use and yet powerful enough to allow all necessary database operations. Since the application kernel reflects domain problems, you do not want to spoil it with database specific code. The interface should rather reflect a suitable domain level abstraction of the data. Providing an interface that offers the complete database functionality would mean to re-implement large parts of the database management system. Since this is too expensive, an SQL-style interface is not feasible.

?? *Flexibility versus speed of development:* Using the logical database model as a guideline for database access may be the easiest solution to implement on a short term basis. However, tuning and maintenance forces you to modify the physical database layout frequently. Since we do not want to change the application kernel as well, we need an interface that is independent of the physical database model.

?? *Performance:* Theoretically, the third normal form is best to work with relations³, but the database system will perform very poor if you use third normal form as physical model too.

?? *Mass problems:* A large data model contains hundred or more entities. Manually writing wrappers or embedded SQL-code for hundreds of entities is a boring and expensive task. Boring tasks are always error prone. A generic solution enables you to use macro expansion, generators or templates for database programming.

Solution

Express the interface in terms of the domain's problem space, that is as relational data model. Start at one point (or entity) of the data model and use foreign key relations to navigate to the other points of interest. Construct a directed acyclic graph (DAG) during navigation. Label every node with the entity, attributes of interest and selection predicates. Label every edge with the foreign key you have used for navigation and its cardinality (one to one or one to many).

Structure

Figure 6 shows a graph representation equivalent to the invoice of the left side.

³ see [Dat94, chapter 10] for details

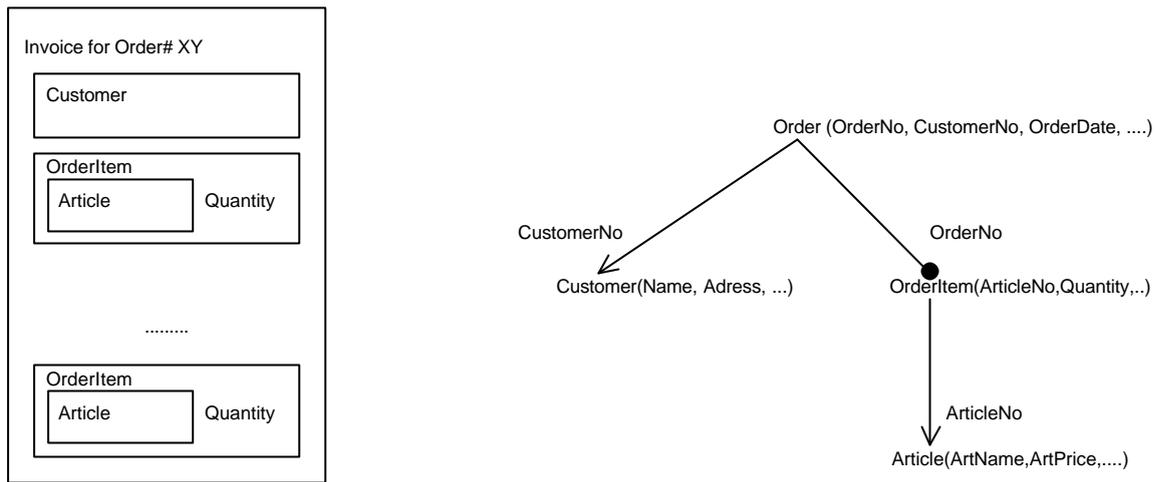
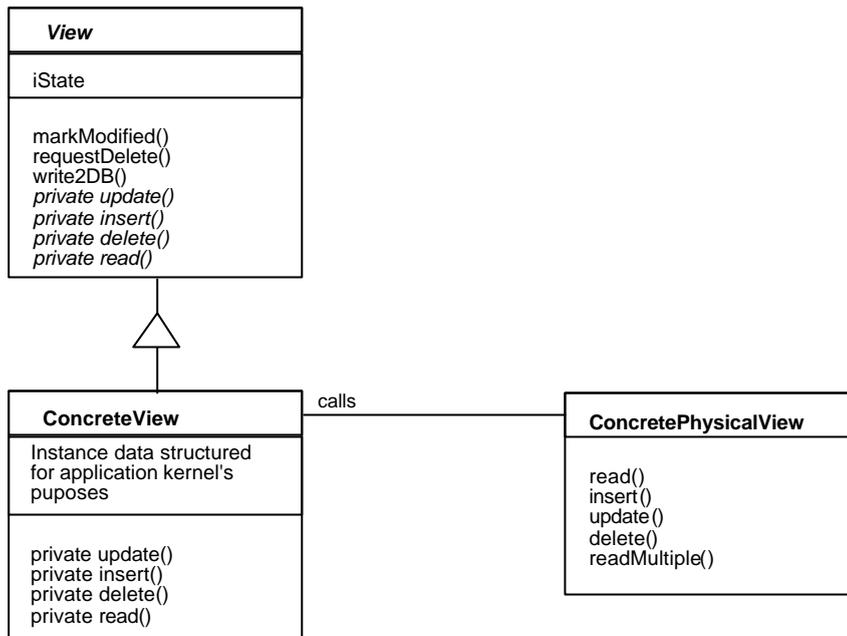


Figure 6: A detail of our order processing system's logical data model. The left side shows the invoice known from Figure 5. The right side shows a DAG-like description of the data contained in the invoice. Every node or leaf of the DAG represents an entity of the logical data model.

To transform this graph into a *Hierarchical View*, write a `ConcreteView` derived from `View`⁴. The `ConcreteView` is the root of the DAG. Define a domain level class for any one of the nodes. Use aggregation to implement *to one* relationships in the graph. Use containers to implement *to many* edges of the graph. A `ConcreteView` constructed this way, fills its domain level attributes from `ConcretePhysicalViews`. The suitable `ConcretePhysicalViews` may be found using hard coded knowledge or the Query Broker.

The database access layer should be able, to treat all `ConcreteViews` uniformly. Therefore, `View` defines their common interface to the other classes of the access layer.

⁴ You may model this approach in 3GL using structures instead of domain level classes. We shall return to this issue in the implementation section.



Implementation

You may define the structure of the *ConcreteViews* using text files or a specialized tool [Würt96]. This allows automatic generation of the *ConcreteViews* for statically typed languages or even run-time definition for dynamically typed languages.

Consequences

- ?? *Inheritance and polymorphism*: The access layer contains no built-in precautions for handling inheritance or polymorphism. Check whether this suits your problem domain.
- ?? *Complexity of the interface*: The interface is minimal because it offers only the basic features the application kernel needs. However, you have to invest effort in generators or templates. There may be a full pay back during tuning but it may as well take several maintenance cycles before you reach break even. Once you have finished the generators, defining new *ConcreteViews* is a matter of minutes.
- ?? *Interface style*: An application using Hierarchical Views follows the structure of the logical data model. The logical data model determines the structure of the code using it. Instead, with an object/relational access layer the object model follows the internal structure of the domain.
- ?? *Ease of use and requirements of the application kernel*: Hierarchical Views reflect only the domain logic while supporting exactly the navigation that the corresponding use cases need. Calling the access layer is simple because the Hierarchical Views encapsulate database specific functions.
- ?? *Performance and flexibility*: Hierarchical Views completely decouple the application kernel from the physical data model. This enables you to tune the database any way you want to without affecting the application kernel's code. The resulting performance gain is much higher than the loss caused by the additional level of indirection the Hierarchical Views introduce.

?? *Mass problems*: At worst you may need more than one `ConcreteView` for every use case. Thus may result in a large number of classes with all resulting problems: Large code, large symbol tables, and so on. However, `ConcreteViews` are generic enough to use templates, macros or code generators.

Example Resolved

Listing 1 shows the declarations of the invoice example, Listing 2 contains the code to process the invoice.

```

struct Customer {
    CustomerKeyType    iCustNumber;
    ... // other properties of the Customer in the logical data model
};
struct Article {
    ArticleNumberType  iArticleNumber;
    ... // Other properties
};
struct OrderItem {
    Article             iArticle;
    QuantityType       iQuantity;
};

class OrderInvoiceView : public View {
public:
    OrderKeyType       iOrder;
    Customer           iCustomer;
    Vector<OrderItem>  iItems; // Any other container will also do
    Money              iSumOfInvoice;
private:
    // private methods you need to obtain data and write data
    // to PhysicalViews

    virtual void update ( void );
    virtual void insert ( void );
    virtual void remove ( void );
    virtual void read ( void );
};

```

Listing 1: The declarations for the invoice example.

The code of Listing 2 is free of database aspects and follows the logical data model. The denormalized physical data model is invisible from the application code. There are only two lines that deal with persistence: The `ViewFactory::getView()` command gets data from the access layer. The `pos->markModified()` method tags the `SumOfInvoice` to write itself back to the database.

```

Void Order::processInvoice (OrderKeyType anOrder) {
    // get the data from the database. We only specify the primary key
    // and leave the rest to the access layer
    OrderInvoiceView * pInvoice =
        (OrderInvoiceView *) ViewFactory::getView( anOrder );
}

```

```

// process invoice items.
ItemIterator itemIter = pInvoice->iItems.begin();
for (; itemIter != iItems.end(); itemIter++) {
    itemIter->iSumOfInvoice +=
        ( itemIter->iQuantity *
          itemIter->iArticle.iArticlePrice );
}

// the view has been changed, so mark it
pInvoice->markModified();
}

```

Listing 2 Implementation of `processInvoice`. The example demonstrates iteration through the items of an order and sums up the prices of all items in the `iSumOfInvoice` property. Note that we traverse two levels of indirection in the logical data model. For reasons of simplicity we omitted the transaction brackets around `Order::processInvoice` as well as some obvious type definitions.

Variants

Many applications are a collection of mostly simple use cases. They need only views with a single level of indirection (like an entity and its dependent entity). In these cases the `ConcretePhysical Views` encapsulate the database access code and provide a sufficiently clean interface to the application, saving the `Query Broker` and the `Hierarchical Views`. However, this variant is not suitable for complex use cases that may touch a two digit number of entities in a single use case. As an example, consider insurance applications.

A more complex variant allows retrieval of historic data. You need this variant if you are not interested only in the current state of a contract but in its state at a given time [Sch96]. To navigate the data model, you have to enrich conditions and navigation edges with expressions for time based navigation [Würt96]. Insurance companies often need this features.

Known Uses

VAA, a standard architecture for German insurance companies, uses this pattern with time navigation [VAA95]. The corresponding *Data Manager Component* is currently under construction. Württembergische Versicherung [Würt96] develops a *Data Manager* using `Hierarchical Views` and a tool to define them.

Many of sd&m's projects have used the simple variant (1:n views) of the `Hierarchical Views` pattern, generating the views [Den91].

Related Patterns

You may use a `Query Broker` to decouple `Hierarchical Views` from the underlying physical database.

Use a `View Cache` to avoid multiple database accesses for the same physical data.

Pattern: Physical View

Context

You have decided to use the Relational Database Access Layer. You use Hierarchical Views as interface to the application kernel and you have chosen, not to incorporate database access into the `ConcreteViews`.

Problem

How do you provide an easy-to-use interface to your physical database tables?

Forces

?? *Simplicity versus Performance*: To achieve good performance you have to optimize your physical table layout using Denormalization, Controlled Redundancy, or Overflow Tables. However, unshielded use of these techniques messes up code dealing with the physical data structure and makes database access complex. Especially Overflow Tables result in intricate code. Despite of these complex optimizations you want to have an easy-to-use interface and maintainable classes.

?? *Flexibility*: Most databases offer you the choice to use either static or dynamic SQL. Because the database pre-compiles and pre-optimizes static SQL queries, it often reduces server load and yields better performance. Some database administrators allow only static SQL on their servers. On the other hand dynamic SQL is more flexible and easily adapts to changes in the database scheme. It is easier to use during development. To satisfy high performance requirements, you may even want to use a low level database API. Higher levels of the access layer should not be aware of these considerations.

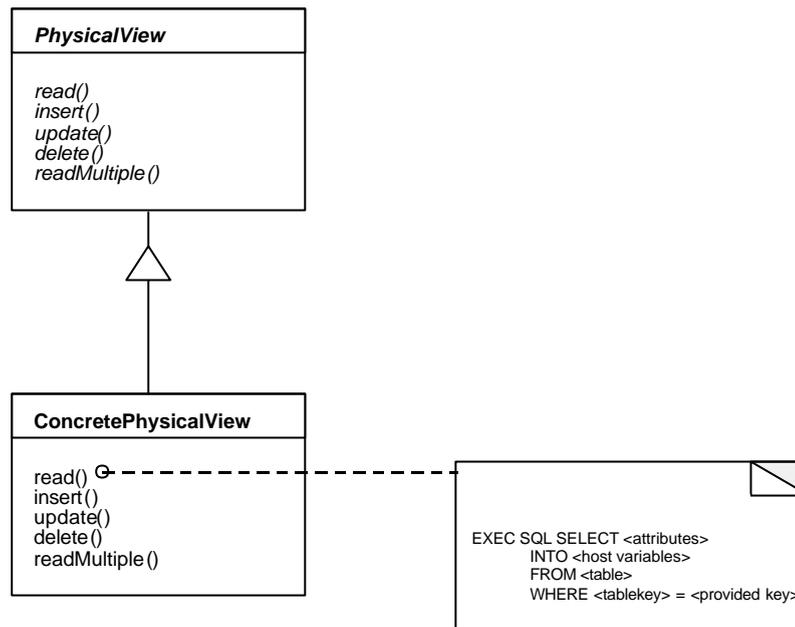
?? *Mass problems and cost of development*: Since large databases may have hundreds of tables you may desire an automatic process to build the interface. Macros, generators or templates reduce the effort. However, a generic interface takes more design effort to develop because it has to cover all possible cases.

Solution

Encapsulate every table and every view with a `ConcretePhysicalView`. Use these classes to handle Overflow Tables too. To provide a uniform interface derive `ConcretePhysicalViews` from `PhysicalViews`.

`ConcretePhysicalView` use record structures like `PhysicalViews` to store their instance data. The main difference is that they shield a single physical table or view instead of multiple physical structures.

Structure



Implementation

What to encapsulate? Each `ConcretePhysicalView` should encapsulate a single SQL statement and its corresponding Overflow Tables. Since Hierarchical Views refer to more than one `ConcretePhysicalView`, you have the choice whether to join two tables on the database using SQL, or whether to join them in the access layer. A good point to start is to define a `ConcretePhysicalView` for every “root table” such as `Customer` and `Article`. Furthermore build one `ConcretePhysicalView` for every “compound entity”, you have defined database views for, such as the `Order/OrderItem` relation. If you use a Query Broker you may analyze its decisions to find further candidates.

Encapsulating read-only views: To keep the Physical Views as simple as possible you should consider, which `ConcretePhysicalViews` have the right to update the data they have read. Physical Views represent database views and most databases do not support writing to views. Hence, if multiple tables are involved, a Physical View with read-only access is simpler than one with read-and-write access. A good idea is to start with exactly one Physical View having *write* access to a certain table.

Programming Tools: `ConcretePhysicalViews` are generic. Use a generator or macro technique to implement them. You may also consider templates.

Use of stored procedures and other APIs: Most databases offer stored procedures to do computation on the database server. Since Physical Views work directly on the database you may implement them with stored procedures or any other API the database offers to access tuples. With this solution you may write tricky optimizations like Overflow Tables in database code instead of a host language plus embedded SQL. However, you put extra load onto the database server and you have to ensure, that all applications comply to this architecture.

Consequences

- ?? *Simplicity*: Physical Views hide the complexity of optimizations and database programming. Because they have no other responsibilities, they are easy to implement. Still, the extra layer adds additional classes. If you plan to omit the Query Broker for hard-wired connections to the ConcreteViews, you should consider carefully, whether it is easier to add the layer or whether the ConcreteViews should do the database access themselves. The latter results in less classes but also less flexibility.
- ?? *Flexibility*: Since Physical Views encapsulate database code, it is their choice, what API they use to access the database. You may have separate sets of classes using different database APIs.. If you want to experiment with different access techniques during runtime, you may even use a Bridge [GOF95] and switch access modes on the fly.
- ?? *Encapsulation*: Physical Views enable you to optimize the physical database structure without affecting upper layers. This simplifies tuning and results in better performance. The penalty of an additional level of indirection is negligible.
- ?? *Mass problems*: It is easy to design a generator that builds first-cut versions of ConcretePhysicalViews. As long as you use no Overflow Tables, you just have to wrap the corresponding SQL statement. More sophisticated generators may also handle Overflow Tables.

Variants

If you have a hard-wired connection between ConcreteViews and ConcretePhysicalViews, you may implement ConcretePhysicalViews as methods of the ConcreteViews. However, this solution is less flexible since you are not able to use the same ConcretePhysicalView twice.

You may also use Physical Views to encapsulate non-SQL databases and file systems such as ADABAS, IMS-DB, CODASYL, and VSAM. As we have mentioned before, you may use this variant to build relational applications on top of legacy databases.

Example Resolved

Figure 7 shows the DAG-definitions of two Physical Views that we need for our invoice example. They correspond to the physical database structure but resolve the Overflow Table (see Figure 1). To simplify the OrderPhysicalView, it should grant update access only to the Order and OrderItem data, but not to the article information. There are other Physical Views to change the Article table.



Figure 7: DAG definitions for two `ConcretePhysicalViews`. Note that the `OrderPhysicalView` encapsulates the `Order` table with its overflow table `OrderItemOverflow`, while `CustomerPhysicalView` encapsulates the `Customer` table alone. See Figure 1 for the physical table structure.

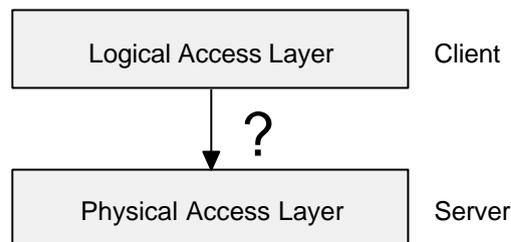
Related Patterns

For a further discussion of optimization, check [Kel+96b]: The Overflow Table pattern describes in detail, how to partially merge tables. Controlled Redundancy contains a discussion on when to grant write access. Narrow View gives hints on Physical Views to select data.

Pattern: Query Broker

Example

Consider the previous example of an invoice. The `OrderInvoiceView` models the logical data structure while the `OrderPhysicalView` and the `CustomerPhysicalView` model the corresponding physical tables.



Context

You have decided to use the Relational Database Access Layer. You use Hierarchical Views as interface to the application kernel and Physical Views to encapsulate database access.

Problem

How to connect the Hierarchical Views and the Physical Views for reading and writing?

Forces

?? *Cost versus flexibility*: The cheapest way to connect two layers is hard-wired coupling via function calls: A `ConcreteView` knows which `PhysicalViews` it has to call. You can generate the corresponding calls using compact table descriptions. This works fine as long as both layers are stable. However, if one layer is unstable you should use some form of decoupling. In the access layer we have the stable Hierarchical Views on top of an unstable Physical View layer. If the system is small enough, you may use a program generator to couple both layers. Still, this approach will produce extensive costs in terms of compilation and software distribution if the system lives for several years. Consider you have to distribute megabytes of database access software to thousands of clients for every change in the physical database model.

?? *Reusability*: Though the Physical Views may change rapidly, they reflect the physical structure of the database. Therefore it is likely that several applications use the same Physical Views but different Hierarchical Views. Writing a separate coupling mechanism for every single application nullifies the gains you get from reusing the Physical Views.

?? *Complexity*: Since the hard-coded solution is not flexible enough, you need a more complex solution. However, extra complexity makes the system more expensive and harder to maintain again.

Solution

Use a Broker [Bus+96] to connect the layers. The Hierarchical Views form the client side of the *Query Broker*, the Physical Views constitute the server side. Describe services using directed acyclic graphs (DAG) and use a tree matching algorithm to find best matches. Let the Query Broker assemble the Physical Views and deliver the result in a Query Result container.

Structure

A Broker is a standard technique for decoupling. Use the standard structure and adapt it to the Database Access Layer framework:

?? the most significant difference between Query Broker and a standard broker is that it usually takes more than one server to handle a request. The mapping to servers is not *one to one* but *one to many*.

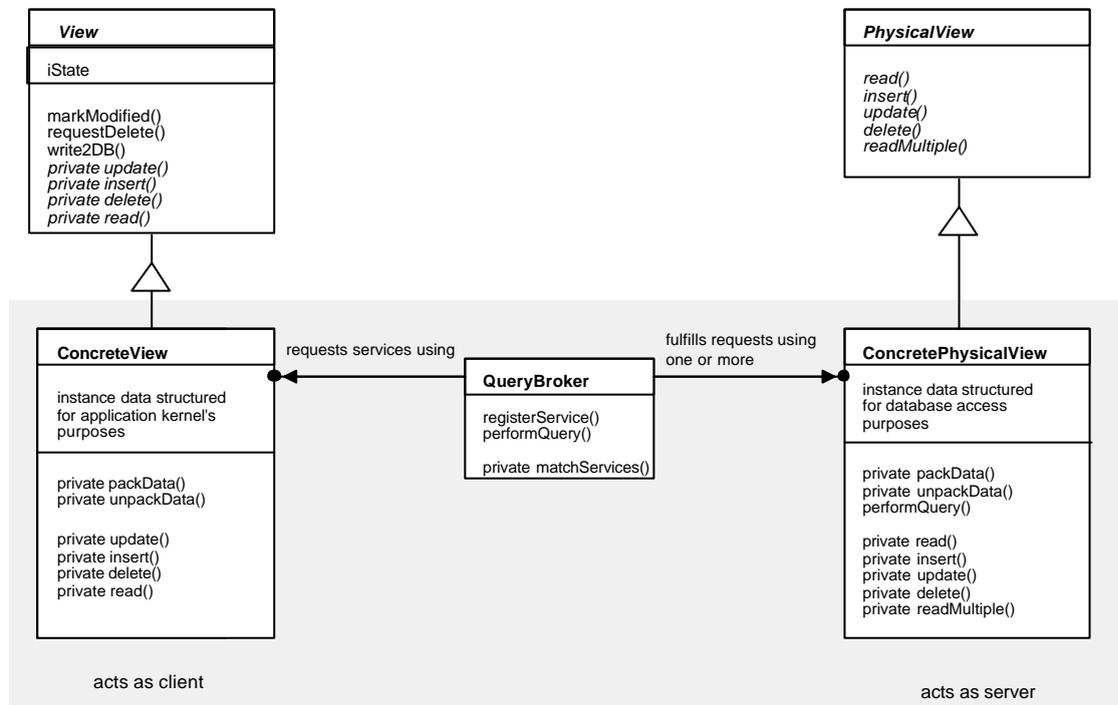


Figure 8: The Query Broker is a Broker adapted to the Database Access Layer Framework.

?? usually Brokers use symbolic names to identify services. As we have a 1:n relation between service requests and servers that fulfill them, this is not appropriate here. Therefore the Query Broker uses semantic descriptions (DAGs) to describe requested views. Consider Figure 9. The left side shows the description of the request for the OrderInvoiceView. The right side depicts the corresponding services. To assemble the ConcretePhysicalViews the Query Broker matches the keys, tagged with white ellipses.

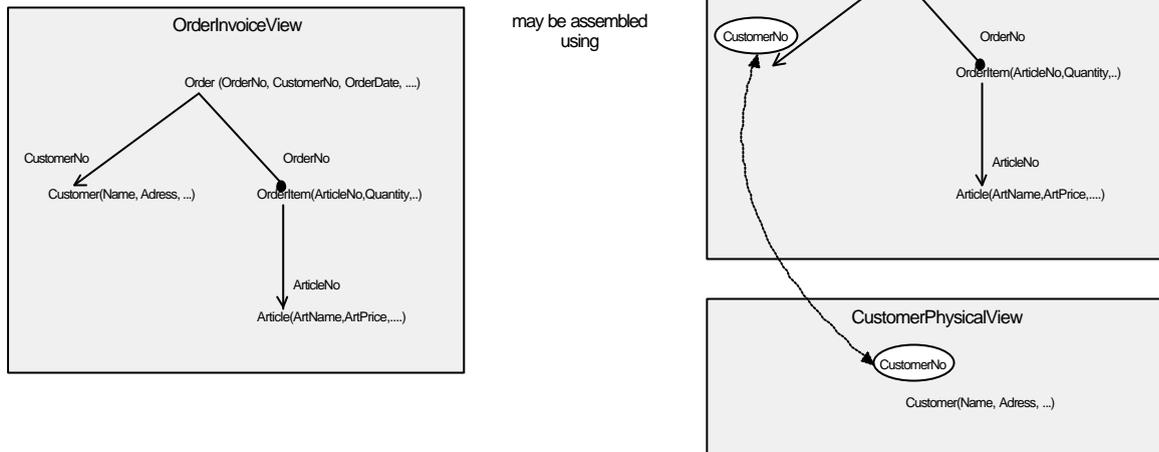


Figure 9: *Tree matching to resolve requests to the Query Broker.* The left side shows the OrderInvoiceView of our order processing system, the right side depicts the corresponding two Physical Views. If the Query Broker gets the left presentation as request, it figures out that the Physical Views on the right side are the best way to satisfy the request.

Dynamic Behavior

In the following scenario an application kernel object creates an OrderInvoiceView causing a database read(). The QueryBroker handles the read() and matches a view description against available services via the matchServices() method. The QueryBroker forwards the request to two different ConcretePhysicalViews: the OrderPhysicalView and the CustomerPhysicalView. These two read() the data from the database and deliver the results by packing the data into result containers. The Broker has to merge both result containers to deliver one result to the OrderInvoiceView. The OrderInvoiceView unpacks the data into its instance variables.

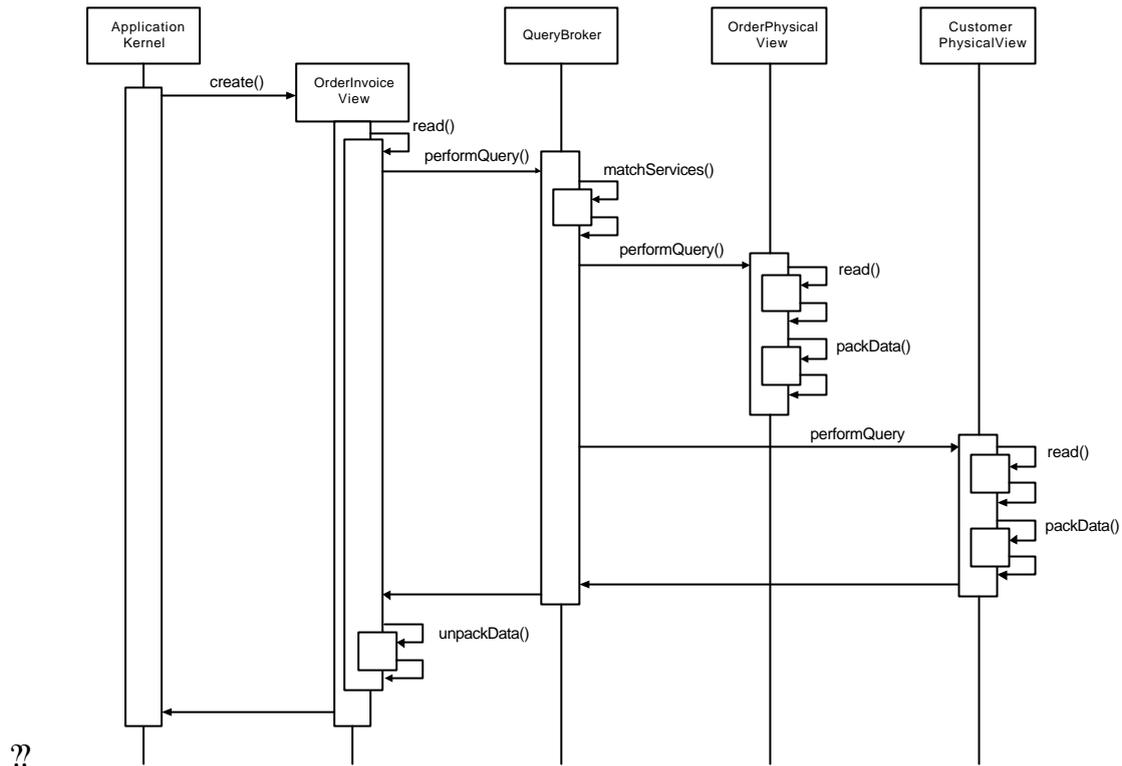


Figure 10: Retrieving data via a Query Broker.

Implementation

?? *Server Registration:* All `ConcretePhysicalViews` have to register at the `QueryBroker` prior to the first database access. You have to take care for it during system initialization. You may use a runtime dictionary, some other form of registry or language specific initialization techniques.

?? *Responsibility for Casting Data Types:* There are two choices, where to cast raw database types to application data types and vice versa. For example, you have to convert a `CHAR(20)` into an `OrderKeyType` and vice versa. You may do the casting the `ConcreteViews` as well as to the `PhysicalViews`. A runtime dictionary may support both alternatives, if it contains the logical data model and the knowledge which attributes to cast into which application data types.

?? *Tree matching:* Matching the DAGs is similar to code generation in compilers, where you have to find good assembler code for a program. So you may use the corresponding algorithms [Aho+86, chapter 9.2]. The Broker may even find several query plans for a request, differing in speed. The matching algorithm has to deal with ambiguous derivations and has to find the fastest solution. You have similar problems in optimizing compilers, which deal with ambiguous grammars for code generation purposes [Kel91].

?? *Query descriptions and result containers:* You have to find a good representation for the DAGs, used to describe queries. On one hand the Views should be able to specify their requests and services easily. On the other hand the presentation should conform to the requirements of the matching algorithm. An easy-to-parse textual presentation is a good choice.

Consequences

- ?? *Flexibility*: The Query Broker completely decouples the Hierarchical and the Physical Views. At runtime, new Physical Views may register and the associations to Hierarchical Views may change.
- ?? *Complexity*: The tree structured result containers, the request descriptions, and a tree matching algorithm make the Query Broker complex to design. However, the Broker is well encapsulated, restricting complexity to a single subsystem.
- ?? *Reusability*: Since the Query Broker is independent from the views it connects, you may implement it as part of a framework. This is even better than reusing only the Physical Views or generators.
- ?? *Cost*: The complexity of the Query Broker makes it expensive to implement. A run time dictionary increases the cost. Implemented in a reusable framework and used in more than one application, the Query Broker will pay off. Hard-wired coupling is cheaper to build but makes optimizations more expensive and causes nightmares when you think about software distribution among several thousand client sites.

Variants

If you use only dynamic SQL, the Query Broker may assemble the SQL statements without using Physical Views. This works fine for clean database models but becomes hard with Overflow Tables. You can this approach in early phases of development. Later, you add more and more Physical View servers using static SQL or other APIs. The Broker shields the application kernel from these changes. It always responds with the fastest service it finds in its registry.

Known Uses

The *Query Broker* is a compilation of various best practices:

The VAA data manager [VAA95] defines views in terms of the logical data model. It uses a generated hard wired coupling of layers. Our experiences at HYPO-Bank [Kel+96a, Col+96] taught us to use dynamic descriptions wherever possible. Two projects at sd&m used other important parts of the approach.

sd&m's LSM project used dynamic SQL, migrated to static SQL and ended up with a tuple interface. The idea was born from bad experiences with a slow database server. The project makes extensive use of a runtime data dictionary and bridges dynamic queries and pre-compiled queries completely.

The Fall/OK project for the German police uses tree matching. The software copes with queries by example on a large data model. The data model changes very rapidly.

CORBA Persistent Object Service [Ses96] also use a Broker. Application kernel objects write their instance data to streams and a Broker (Persistent Object Manager) forwards the stream to some Persistent Object Service (database or other). Persistent Object Services may be arbitrary data-

bases not known to the object. This is also the simple a case of a *one to one* mapping between service requests and servers that fulfill them.

Related Patterns

[Bus+96] contains a comprehensive discussion of Brokers in general.

Brown and Whitenack describe a Broker [Bro+96] on a per class basis. Note, that Query Broker is more general.

Acknowledgments

We would like to thank *Frank Buschmann*, our PLoP shepherd, for his great advice and support. We are also obliged to the participants of the BOF workshop at EuroPLoP '96 and to the member of the PLoP '96 writers workshop..

Our colleagues *Andreas Mieth*, *Uli Zeh*, and *Andreas Wittkowski* contributed their profound database knowledge during the review process. Wolfgang got many valuable insights from the VAA Data Manager Group and the VAA Architecture Board. Special thanks to *Johannes Schlattmann* (LVM Versicherungen), *Hans Hoffmann* and *Gerhard Pallauro* (Württembergische Versicherung), *Volker Bohn* and *Klaus-Walter Müller* (Siemens Nixdorf Information Systems). Finally we thank the German Ministry for Research and Technology for funding our project under contract name ENTSTAND

References

- [Aho+86] **Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman:** *Compilers: Principles, Techniques, and Tools*, Addison-Wesley 1986.

- [Bro+96] **Kyle Brown, Bruce G. Whitenack:** *Crossing Chasms, A Pattern Language for Object-RDBMS Integration*, White Paper, Knowledge Systems Corp. 1995. A shortened 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.

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

- [Col97] **Jens Coldewey:** *A Database Access Layer for ODBMS*, In **Akmal Chaudri, Mary Loomis (Eds.)**, *Experiences with ODBMS* [working title], Prentice Hall 1997 (to appear).

- [Dat94] **Chris J. Date:** *An Introduction to Database Systems*, Sixth Edition; Addison-Wesley 1994

- [Den91] **Ernst Denert:** *Software-Engineering*, Springer Verlag 1991.
- [GOF95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:** *Design Patterns, Elements of Reusable Object-oriented Software*, Addison-Wesley 1995.
- [Gra+93] **Jim Gray, Andreas Reuter:** *Transaction Processing, Concepts and Techniques*, Morgan Kaufmann Publishers 1993.
- [Kel91] **Wolfgang Keller:** *Automated Generation of Code using Backtracking Parsers for Attribute Grammars*, ACM Sigplan Notices, Vol. 26(2), 1991.
- [Kel+96a] **Wolfgang Keller, Christian Mitterbauer, Klaus Wagner:** *Objektorientierte Datenintegration über mehrere Technologiegenerationen*, Proceedings ONLINE, Kongress VI, Hamburg 1996.
- [Kel+96b] **Wolfgang Keller, Jens Coldewey:** *Relational Database Access Layers: A Pattern Language*, Proceedings PLoP '96, Allerton Park 1996.
- [Lan96] **Manfred Lange:** *Abstract Constructor*, Preliminary Conference Proceedings EuroPLoP, First European Conference on Pattern Languages of Programming, Irrsee, Germany, 1996.
- [Mar95] **Robert C. Martin:** *Designing Object-Oriented Applications Using the Booch Method*; Prentice-Hall International, London, 1996
- [ODMG96] **Rick G. G. Cattell (Ed.) et. al.:** *Object Database Standard: ODMG-93 - Release 1.2* Morgan Kaufmann Publishers, San Mateo, California, 1996.
- [Rum+91] **James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen:** *Object-Oriented Modelling and Design*, Prentice Hall, 1991.
- [Sch96] **Johannes Schlattmann:** *Die Anwendungsarchitektur der Versicherungswirtschaft*, Historienkonzept, Entwurf, GDV Bonn, 1996.
- [Ses96] **Roger Sessions:** *Object Persistence, Beyond Object Oriented Databases*, Prentice Hall 1996
- [VAA95] **GDV:** *VAA - Die Versicherungs-Anwendungs-Architektur*, 1. Auflage, GDV, Bonn 1995
- [Würt96] **Württembergische Versicherung:** *Projekt Datenmanager*, private communications 1995 - 1996.
- [Wit96] **Andreas Wittkowski:** *Datenbankdesign & Performance*, slide presentation, sd&m Internal Lecture Series, 1996.