Relational Database Access Layers
A Pattern Language

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

---

The Relational Database Access Layer Pattern Language provides a uniform mechanism for efficient database accesses and encapsulation of database system aspects. The pattern applies as a persistence mechanism for data abstraction modules.

---

1 Motivation

Imagine you are building an order processing system. Domain objects like Order, OrderPosition and Customer can well be expressed in terms of the relational model. Typical Business Information Systems can be designed using only a small amount of inheritance relations or complex relationships between objects. They normally consist of 30 or more domain entities and provide a complex application kernel. Figure 1 shows a small fraction of the logical and physical data models of an order processing system and also the structure of an invoice. We will use the example of „Printing an Invoice“ as our running example throughout this pattern language.

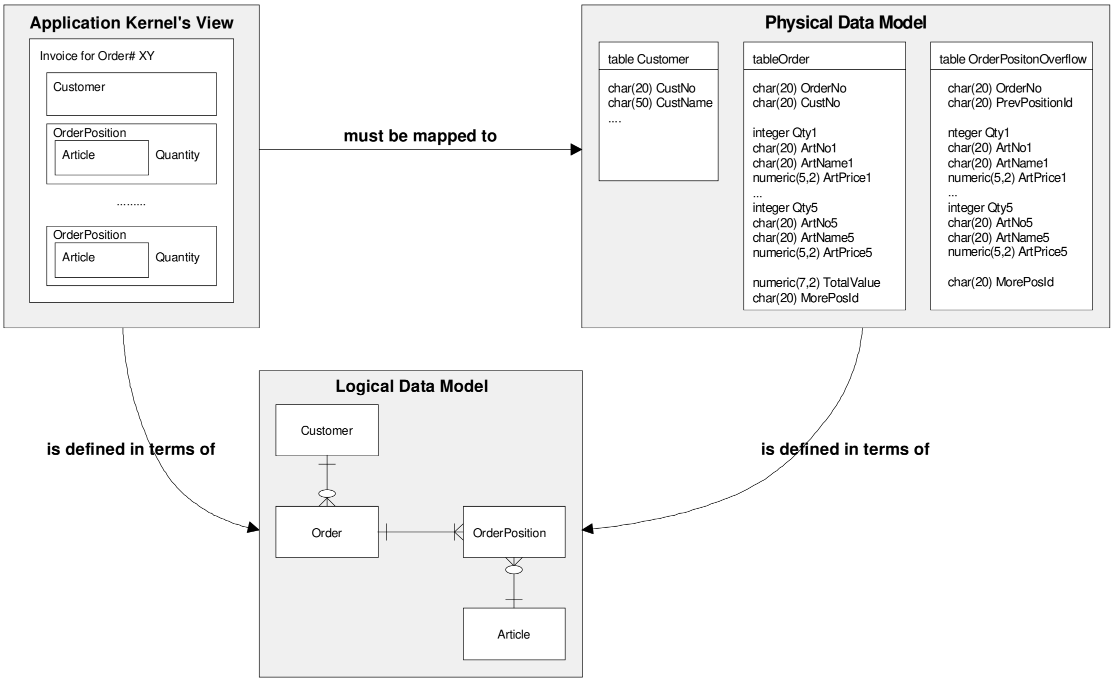


Figure 1: Part of an Order Processing System

Now imagine you were the designer of the above order processing system. As a reasonable first cut approach for implementing a physical data model you might use the logical data model. Logical data models are usually specified in third normal form [Dat94]. The system works fine in a single user test but after some time, after more and more of your company's clerks use the system, performance problems appear.

When measuring the system's performance you will probably detect many superfluous database operations. You might also find database operations that are slow due to joins or moving unnecessary large amounts of data at a time.

To provide better performance you may denormalize the physical data model. A statistical analysis of the database's contents yields that 90% of the Orders have no more than five OrderPositions. The physical data model is then optimized by putting the first five OrderPositions into the Order table and by creating an OrderPositionOverflow table for the 10% cases that have more than five positions (see the above Figure). Controlled Redundancy may further be used to integrate the Article entities ArtPrice and ArtName columns into the Order table. This design allows the order processing system to read 90% of all orders with two single page database accesses. One for the Order table and another one for the Customer table. There is no need for a single join operation.

But what happens to the application code and embedded SQL-statements? There are embedded SQL-statements amidst the application kernel code. Thus you must check and rewrite large portions of the application kernel code to adapt it to new table schemes. Handling overflow tables lets the SQL code explode. This procedure must be repeated for each improvement of the database structure.

Relational Database Access Layer combines a framework with a pattern language for dealing with applications that use their databases in a relational fashion. The pattern language is driven by three main forces: Performance, Flexibility and Decoupling of the application kernel from the underlying database.

2 Pattern Language Map

We divided the pattern language into three Parts. Chapter 3 explains the „Relational Database Access Layer" framework. The framework defines roles and responsibilities for its components. Chapter 4 contains a collection of patterns that are needed to implement the framework.

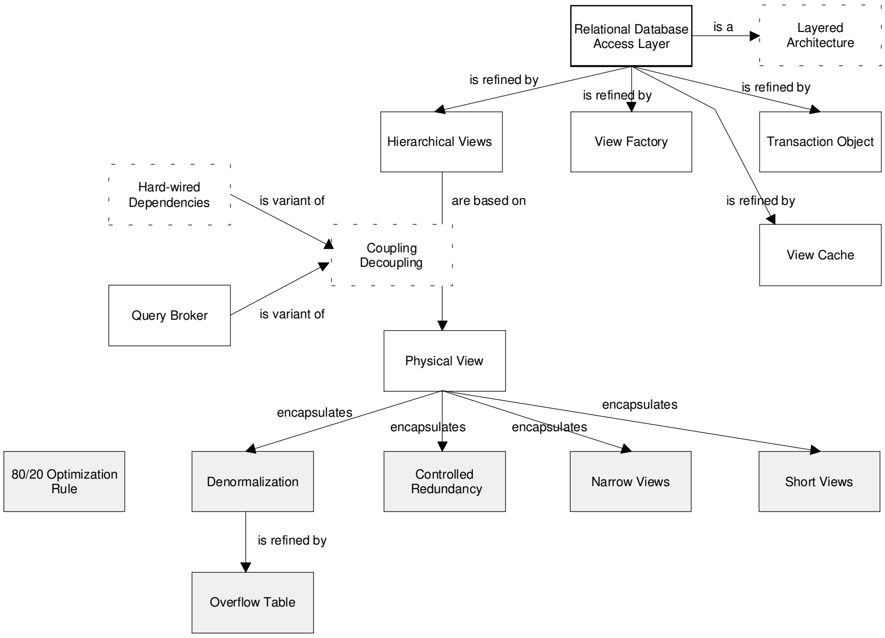


Figure 2: A Map of the pattern language. Dotted boxes are known design principles. Unshaded boxes implement the „Database Access Layer Framework". Shaded boxes are database optimization patterns.

Chapter 5 contains a collection of database optimization patterns. These patterns may be applied by a developer with the aim to speed up a relational database application. The 80/20 Optimization Rule is a general problem solving strategy that may be applied in many situation.

Notational Conventions We use OMT [Rum+91] for object diagrams. For object interaction diagrams we use the notation used in the GOF Book [GOF95]. Related Patterns are referenced as ⇒ *Another Pattern*. Patterns references, that are not resolved in this language, will be followed by a citation like [GOF95]. Code examples are written in C++.

Acknowledgments We would like to express our thanks to our PLoP shepherd Frank Buschmann who gave more than great advice. We'd also like to thank the participants of a spontaneous BOF workshop at EuroPLoP 1996. This resulted in splitting a single pattern into the pattern language presented now. Our colleagues Andreas Mieth and Uli Zeh helped improve this paper by reviewing it until they considered it at least a significant part of sd&m's knowledge concerning the field. They will continue to do so. The Database Optimization Patterns were mined from a lecture by our colleague Andreas Wittkowski [Wit96]. Wolfgang got many valuable insights from working with the VAA Data Manager Group and the VAA Architecture Board. Special thanks go Johannes Schlattmann (LVM Versicherungen), Hans Hoffmann and Gerhard Pallauro (Württembergische Versicherung), Volker Bohn and Klaus-Walter Müller (Siemens-Nixdorf Information Systems). Finally we would like to thank the German Ministry for Research and Technology for their funding of our project under contract name ENTSTAND

State of the Language This pattern language is a beginning to cover the field of access to relational databases. There's a whole bag of tricks left for later versions. Relational database access by batches will be given more attention in future versions. There are more optimization patterns that make a good database administrator and there are more variants of the framework. Some aspects we have listed as implementation issues can also be described in more detail in future work. Some side comments may be documented as own patterns. But you have to start with something.

3 Framework Pattern „Relational Database Access Layer“

Pattern: Relational Database Access Layer

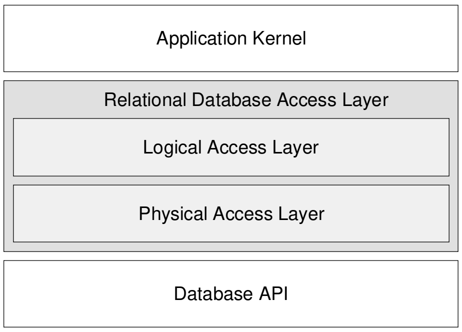
Context You are writing a Business Information System like the above order processing system. The relational calculus is an appropriate representation of the domain logic. The resulting data model is simple and uses inheritance only sparingly. Maybe you have to use a relational database for reasons beyond the project's scope. The effort of mapping the relational model to an object-oriented representation is high compared to the gains.

Forces The three most important aspects when dealing with access to relational databases are performance, flexibility and decoupling the application kernel from the underlying physical data model.

The application kernel should refer to a stable interface expressed in terms of the logical data model. As to support performance optimization and application code stability at the same time, we need to be able to flexibly change the physical database layout without changes to the views on the logical data model. The application kernel should remain stable and should hence be decoupled from the fast changing physical data model.

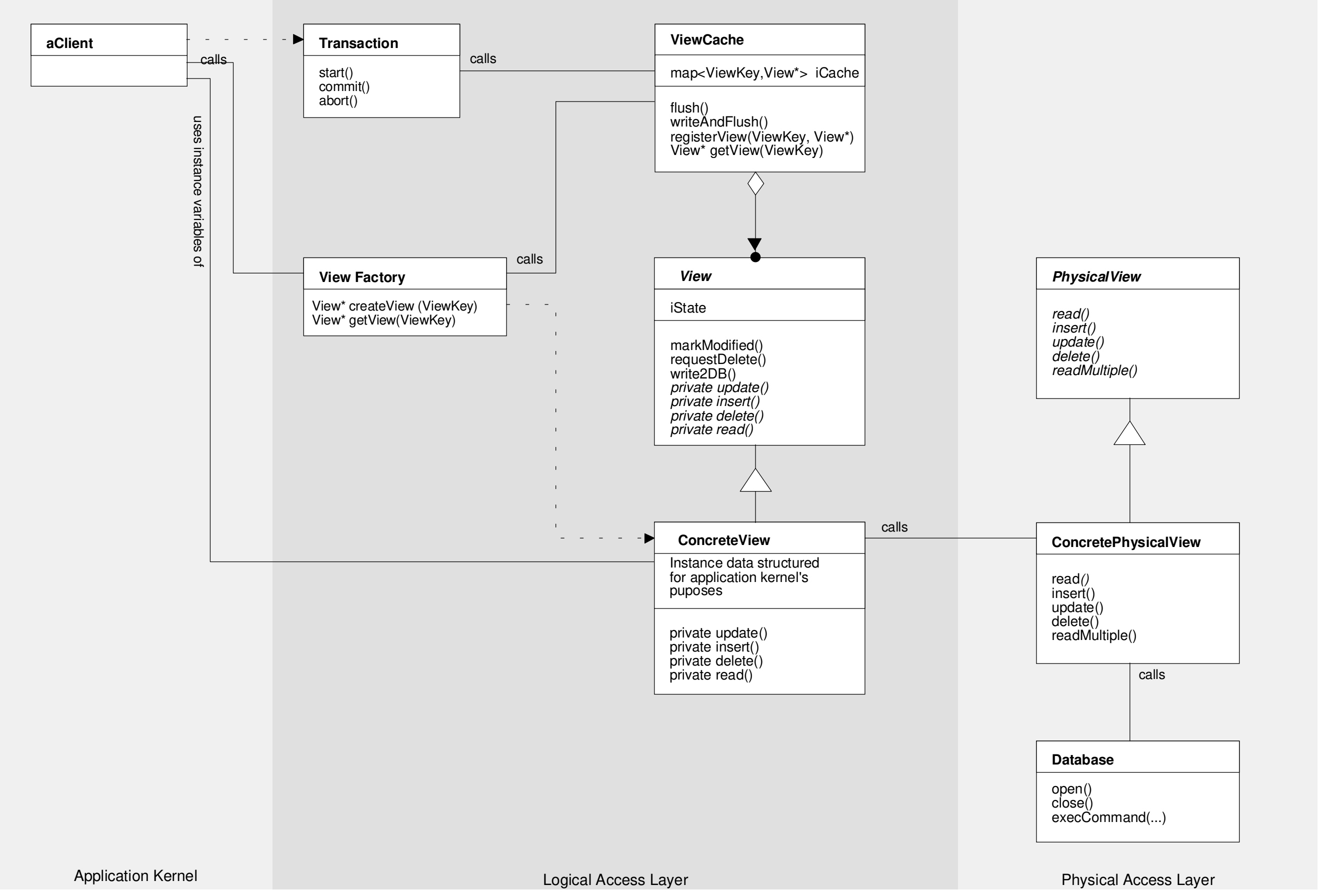
You can improve performance by an array of patterns like prefetching larger chunks of data, caching, or table optimizations (see ⇒ *Chapters 4 and 5*). A Database Access Layer should allow such mechanisms to be integrated without affecting the application's code. To further trick the performance/flexibility tradeoff, it is desirable to be able to use static queries and dynamic SQL in the same access layer, also without affecting application kernel code.

Problem How do you manage access to a relational database?



Solution Use a layered architecture consisting of two layers. The application kernel's interface, called Logical Access Layer, remains stable. The access layer to the physical database, called Physical Access Layer, may be adapted to changing performance needs. Use a decoupling mechanism like a broker to couple the two layers.

Structure The Logical Access Layer provides the stable interface for the Application Kernel. The Physical Access Layer offers database access. This is encapsulated by two mechanisms. The database engine is encapsulated by a Database class. ConcretePhysicalViews provide an abstraction on top of one or more physical database tables. We did not show the coupling of the two layers in more detail here. The coupling may be hard-wired. A more flexible way to couple the Logical and Physical Access Layers is using the ⇒ *Query Broker* pattern.



Participants

Transaction

- Offers an interface that allows start(), commit() and rollback() of transactions.
- Will be reinstantiated for each new transaction and is typically destroyed after commit() or abort(). The usage is similar to that of an [ODMG96] transaction object. See the ⇒ *Transaction Object* pattern for a detailed discussion.

ViewFactory

- Is responsible for the activation and creation of Views when they are loaded by the application kernel.
- The ViewFactory uses a ⇒ *View Cache* to check whether a requested ConcreteView has already been loaded and loads new ConcreteViews only if they are not already cached in the transaction running.
- The key for ConcreteViews is any key or selection criterion. An abstract key class ViewKey is required.
- The ViewFactory is a ⇒ *Singleton* [GOF95] or static object that is responsible for all of the View's subclasses.
- References to Views can only be obtained by asking the ViewFactory with a getView() or createView() method.
- For a further discussion see the ⇒ *View Factory* pattern.

ConcreteView

- Is a ⇒ *Hierarchical View* upon the logical data model. Hence there must exist different ConcreteView classes, each of them suited for one or more use cases of the application kernel. Members of ConcreteViews are application data types, not raw database types. Application data types are typically used at application kernel level instead of raw database data types[1]. They have additional methods to check their contents, reformat it, or format it for output.
- Knows how to write itself to the Database using one or more 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. The suitable ConcretePhysicalViews that are needed to perform the operation are determined using hard coded knowledge or better a ⇒ *Query Broker*.
- Is able to fill itself with data from the Database using ConcretePhysicalViews. Depending on the decoupling method used, this may include joining data that may be physically delivered by several ConcretePhysicalViews according to mapping information provided by hard-coding or better a ⇒ *Query Broker*.

ViewCache

- Keyed Container of Views that forms the access layer's cache mechanism.
- At commit time the ViewCache's writeAndFlush() method sends a write2DB() message to all Views it contains in order to write them to the database if they had been modified. If the transaction is aborted, the Transaction object calls the flush() method to clear the cache instead.
- See the ⇒ *View Cache* pattern for details.

View

- Defines a uniform protocol for ConcreteViews for convenient use by the ViewCache. The protocol uses abstract methods, that must be redefined by the View's descendants in order to really manipulate the underlying physical database.
- Can be informed that it has been modified using the markModified() method. This will cause a database update if the current transaction is committed.
- Its descendant's physical database equivalent can be deleted by the requestDelete() method that will cause a delete on the database at the end of the transaction. Be careful not to confuse this method with a destructor. While the destructor instantly eliminates the objects from memory, requestDelete() deletes the object from the database (and from memory) at the end of the transaction. To avoid

---

[1] Application Data Types may be discussed as an own pattern later.

dangling references, the requestDelete() method should be the only accessible way to delete database records.

PhysicalView

- Defines a uniform protocol for ConcretePhysicalViews. The protocol uses abstract methods, that must be redefined by the PhysicalView's descendants in order to really manipulate the underlying physical database via the Database class.
- Bundles database access functions, encapsulates database behavior and translates database error codes into exceptions[2].
- See the ⇒ *Physical Views* pattern for details.

ConcretePhysicalView

- One ConcretePhysicalView wraps typically one physical database relation. This might be a physical database table or a view. If the database does not support direct update of views, the ConcretePhysicalView is responsible for issuing the appropriate write commands.
- A ConcretePhysicalView may also wrap more than one physical database tables in case so called ⇒ *Overflow Tables* have been used in a process of ⇒ *Denormalization* of the database.

Database

- Encapsulates the database engine. Provides methods for starting database connections, issuing database commands and receiving results.

Dynamic Behavior Dynamic behavior of the Database Access Layer will be discussed in the patterns that are used to implement different aspects of the Database Access Layer Framework (see ⇒ *Chapter 4*).

Implementation There are several implementation issues to be considered:

- *Treatment of Mass Updates:* Mass updates are statements compliant to the form „update .. where" that manipulate a larger set of records with a single database query. These statements are not easily incorporated in the ⇒ *View Cache* pattern given here. Incorporating mass updates would mean: Perform a mass read into the ViewCache, manipulate single records, and write them back one by one into the database. Such a solution will be slower than performing the same task directly on the database. This is why mass updates should be treated as separate functions of ConcretePhysicalViews.
- *Batches* will need specialized treatment. There is an array of patterns that deal with batch database access. These are waiting to be mined.
- *Multiple Read Queries*: We have mostly skipped multiple read queries in the above. More information is contained in the ⇒ *Short Views* and ⇒ *Narrow Views* patterns.
- *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). Mass read operations are typically used to fill list boxes. They have the typical form „select <fields> from ... where". Integrating them into a consistency checking scheme at commit time would mean rereading all read records of a transaction at commit time and comparing them to their before images. If just a single record differs that was ever used to fill a list box (see ⇒ *Short Views*) the transaction has to be aborted. This is not only a serious threat to

---

[2] The term *exception* is used here as common in object oriented languages. Most object oriented libraries will throw an exception for unwanted situations like access to an empty container. In 3GL languages this would be handled as a NULL Pointer return value, as the situation is undesired but *expected*. In object oriented languages exceptions are usually not mixed with return codes. Exceptions are not reserved for severe system errors (undesired *and unexpected* behavior) but may also be used in situations where a return code would be appropriate in a 3GL language.

performance - it also doesn't add any value to consistency. In most cases records that were used to fill list boxes do not play any role that could flaw the consistency of a task performed by a data abstraction module. This is why it is usually sufficient to use so called „browse“ consistency for data that are not used to compute results during a transaction.

- *ConcretePhysicalView Objects in the Presence of dynamic SQL:* If a relational system supports dynamic SQL without a runtime penalty, the ConcretePhysicalViews may be removed from the pattern and replaced by the ⇒ *Query Broker* that is told which data are required by a ConcreteView and generates a sequence of SQL statements that fetch or write the data. For static SQL the queries are provided by the ConcretePhysicalViews and will in most cases be generated from meta information.
- *Database Connections* should remain established as long as possible. They should live longer than a single transaction as initiating database connections is an extremely expensive task.

Sample Code Sample code for the framework is provided with the scenarios in the various patterns of the language (see ⇒ *Chapter 4*). A general example here would only mean redundancy.

Variants The following variants may be found:

- *Using a session ViewCache:* The session memory provided by the ViewCache class might not be incorporated in a solution if no long transactions or scrolling are needed. For a simple dialog system that supports only the manipulation of exactly one record in one transaction this is a feasible approach. Simple dialog systems are built that way. If the application kernel objects have a notion of transactions that spans more than one record at a time, the session memory is recommended.

  A cache is the natural choice to implement user transactions on top of a conventional transaction monitor[3] (TMs) like IMS, CICS, or UTM. As all changes are written down at commit time, this will typically contain the changes of several dialog steps of a TM. Hence we have a possibility to emulate user transactions that last several dialog steps using a conventional TM.
- *Using non-relational Databases:* The Physical Access Layer may also be used to hide the existence of non-relational databases and file formats, like IMS-DB, CODASYL databases or VSAM, to upper layers. In practice the framework has been used to build relational applications on top of hierarchical or other legacy data.

Known Uses The pattern can be found in the VAA Data Manager in an extended form together with editors for meta data and complex mappings for hierarchical data [VAA95]. The VAA Data Manger is heavily influenced by the Data Manager Architecture of Württembergische Versicherung [Würt96].

Some basic ideas of the whole pattern language have been sketched by Denert [Den91, pp. 230-239]. The pattern has been used in various variants in many of our companies projects that use relational databases. Parts of the scheme have been implemented for Thyssen, Deutsche Bahn, and HYPO Bank [Kel+96] to name a few customers.

Consequences The Relational Database Access Layer Framework provides the following **benefits**:

- *Performance:* There is some run time penalty for mapping and the ConcretePhysicalViews. On the other hand the access layer performs caching and eases tuning, so the database accesses are optimized. Usually this gain is ten or more times larger than the loss. So the access layer will speed up the application considerably.
- *Easy tuning and database maintenance:* If the implementation uses a ⇒ *Query Broker*, instead of hard wired dependencies between ConcreteViews and ConcretePhysicalView, maintenance and tuning is performed by adding new ConcreteViews and ConcretePhysicalView and not by modifying large

---

[3] Here we have yet another pattern called „Long Transactions“ on top of classical transaction monitors. It has already been sketched in [Kel+96].

portions of code that directly refer to the physical database. The application code remains stable while the underlying physical database can be adapted to changing performance requirements.

- *Encapsulation of the database:* A database access layer provides the application kernel with a well encapsulated mechanism for transactions, database access and caching. This will save considerable programming effort for the programmers who are able to concentrate on domain code after the database access layer is written.
- *Access Layers as reengineering strategy:* The access layer can be used by existing applications to provide decoupling of physical and logical data models. This is useful to reengineer mission critical legacy applications and to port them, for example from a hierarchical to a relational database. As a first step a database access layer is inserted into the code. This is a single, technical step with manageable risk. It offers the possibility to separately reengineer the database and the application kernel in succeeding projects.
- *Use of non-relational databases:* The relational database access layer has been used on top of hierarchical and other databases. The kind of database used may be completely hidden in the Physical Access Layer.

The following is a question of applicability and may turn out to be a **benefit** or a **disadvantage**:

- *Relational View upon the database:* Since the access layer does no transformation from a relational to an object-oriented application model, the domain code has to work with the relational view of the data. You should carefully consider whether this matches the application logic or not. It is not a good idea to save the effort for a more complex access layer when the required mapping is still implicitly performed in the application layer.

  This is typical for complex OO applications like case tools or groupware applications on top of relational databases. In the end not a single dollar is saved and the application code is mingled with complex, technical tasks that decrease readability, testability, and maintainability of the domain specific code.

  On the other hand, many comparably simple business applications are relational in their nature and can live very well with a relational view upon data. The Relational Database Access Layer is targeted at such applications.

There are also **drawbacks** of the framework:

- *Cost*: The coding and specification effort for a database access layer depends on a projects requirements. In our practice we have seen access layer projects from 0,5 to 35 person years. The cheaper ones used generators and hard coded dependencies. The luxurious ones defined their own set of tools for defining views in terms of a logical data model. The investment sum you might want to spend on an access layer depends on the size and quantity of projects ahead of you and on the life expectation of your applications.
- *Reengineering of Triggers and Stored Procedures*: The access layer is not easy to use with existing stored procedures or database triggers. As there is a cache integrated into the access layer you will better not try to imitate the effect of stored procedures at the cache level. Using the access layer on top of an existing database that makes heavy use of triggers and stored procedures cannot be recommended. Stored procedures are generally considered a violation of a clean Three Layer Architecture (UI / application kernel / database). For some high-speed applications that stress a relational database to its limits using triggers and stored procedures, the Database Access Layer Framework in the form presented might turn out to be a project killer. If stored procedures are to be used, this has to be decided early in the design effort and will result in modifications to the design of the access layer.

See also On the top level of the framework design we have applied ⇒ *Layers* [Bus+96]. This is refined by using ⇒ *Hierarchical Views* as the main abstraction for the Logical Access Layer and by applying ⇒ *Physical Views* as the main abstraction for the Physical Access Layer. The two layers can be coupled using *hard-wired dependencies* (ordinary calls) or using a ⇒ *Query Broker* for two side anonymization allowing maximum flexibility for database optimizations as described in ⇒ *Chapter 5*. By using separation of concerns we could find the other patterns that make the structure of the framework. These are described in ⇒ *Chapter 4*.

The basic interaction patterns between ViewFactory, ViewCache, Transaction object, and application kernel objects can also be found in Object to Relational Access Layers that map persistent objects to relational databases [Bro+96, Col+96]. The layer may also be used as a base for Object to Relational Access Layers.

4 Structural Patterns

Pattern: Hierarchical Views

Example In the progress of designing the order processing system you want to write a method Order:: printInvoice (OrderKeyType anOrder, char* aFileName). It writes an Invoice for a given order (anOrder) to a file for later display at a terminal or printing. The structure of the invoice may be, for example, as depicted in the upper left corner of Figure 3.

Context Writing code that touches larger portions of the underlying logical data model, you decided to base the application kernel on a layer of views that are independent from the structure of the physical database.

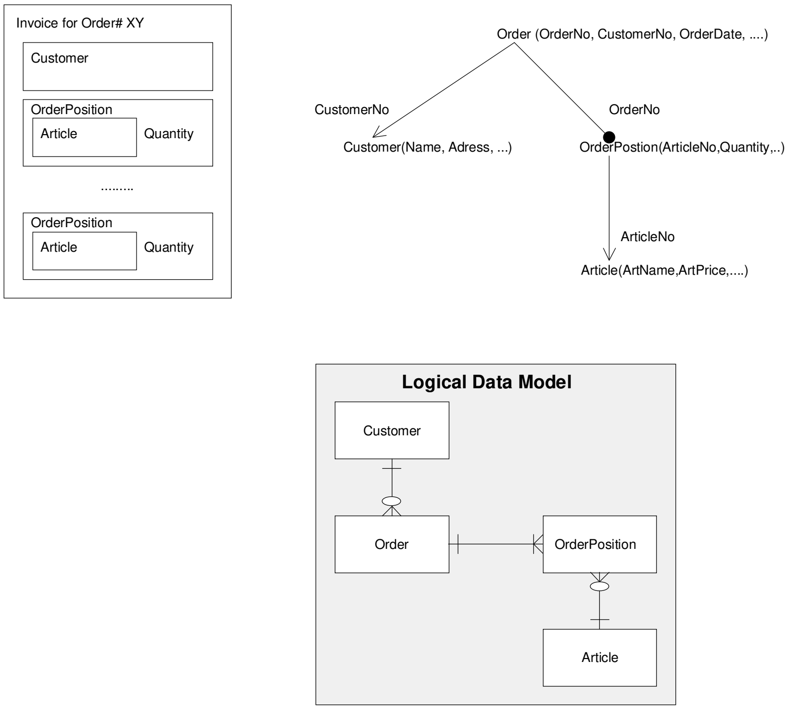


Figure 3: Views Expressed in Terms of a Logical Data Model

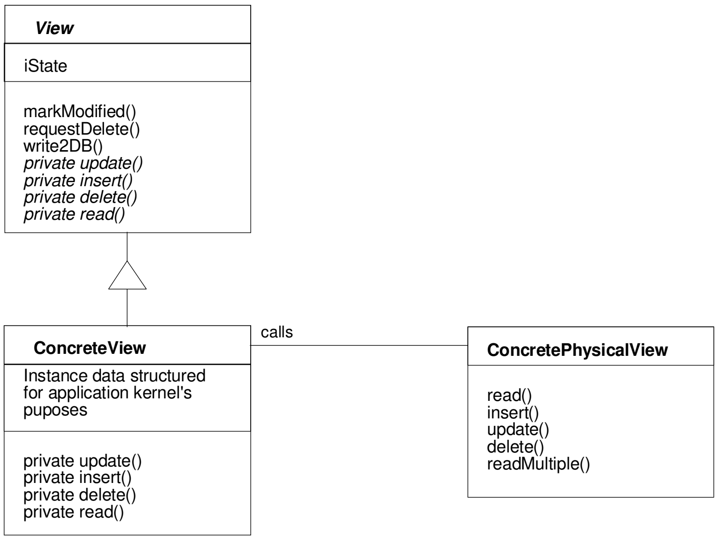
Forces In the process of writing a database application it may happen to you that the physical database changes due to performance considerations while you are still developing new modules. If the interface to an application kernel is expressed in terms of the physical data model, you will loose flexibility as you have to rewrite the interface each time the physical database is changed. Rather you should be able to totally decouple the database interface from the underlying physical database structures. The interface should be simple, easy to understand and should not require large effort to use and create. The interface should also be free of any database specific knowledge. This will allow application kernel programmers to concentrate on domain logic while database experts can concentrate on improving the database.

Problem What will the interface to the database look like?

Solution Use an interface that can be expressed in terms of the domain's problem space. The problem space in our application domain can be modeled in terms of a relational data model. You typically start at one point (or entity) of the data model and use foreign key relations to navigate to the points of interest to you. This can be expressed as a directed acyclic graph (DAG) where each node is labeled with the entity, attributes of interest and selection criteria. Each edge is labeled with the foreign key you use for navigation and an arity (*one to one* or *one to many*).

Structure The upper right corner of Figure 3 shows a graph notation that matches the requirements defined by the solution. This graph can easily be transformed into a recursive structure of records and containers of records. A ConcreteView obtains its data by issuing requests to a ⇒ *Query Broker* or by using hard coded knowledge which ConcretePhysicalViews to call. We will explain the interface needed for that when we discuss the design of the ⇒ *Query Broker*.

As ConcreteViews are aggregated by the ⇒ *ViewCache* we need a protocol that allows us to treat Views as cached objects. We therefore define an abstract class View that provides a uniform protocol for all ConcreteViews and allows keeping them in the ⇒ *ViewCache*.



Implementation To implement the data structure for a ConcreteView, write a class that is derived from View. This class implements the root of a DAG (directed acyclic graph) that describes the needed view. The DAG for our example is depicted in the upper right corner of Figure 3. Use a record for each of the other nodes. Use aggregation to implement *to one* relationships in the graph. Use containers to implement *to many* edges of the graph. In case of our invoice example this results in the following declarations for the class header.

```
struct Customer {
    CustomerKeyType       iCustNumber;
    CustomerNameType      iCustName;
    CustomerAdressType    iCustAdress;
};
struct Article {
    ArticleNumberType     iArticleNumber;
    ArticleNameType       iArticleName;
    Money                 iArticlePrice;
};
struct OrderPosition {
    Article               iArticle;
    QuantityType          iQuantity;
};
// OrderInvoiceView class definition
class OrderInvoiceView : public View {
public:
    OrderInvoiceView ( OrderKeyType anOrder );
    ~OrderInvoiceView ( void );
    // instance variables
    OrderKeyType                            iOrder;
    Customer                                iCustomer;
    Clist<OrderPosition,OrderPosition &>    iPositions;
    Money                                   iSumOfInvoice;
    // private methods that are needed to obtain data and write data
    // to PhysicalViews
private:
    virtual void update ( void );
    virtual void insert ( void );
    virtual void remove ( void );
    virtual void read ( void );
};
```

Example resolved

The resulting code for printing the invoice with the Order::printInvoice method is free of database aspects and follows the above line in the logical data model. The denormalized physical data model is kept invisible from the application code. There are only two lines that deal with persistence. The ViewFactory::getView() command is used to get data from the access layer. The pInvoice.markModified() method states that the SumOfInvoice must be written back to the database.

```
void Order::printInvoice (OrderKeyType anOrder, char* aFileName) {
    // init the output file for the invoice
    ofstream invoice ( aFileName );
    // get the appropriate data
    OrderInvoiceView * pInvoice =
        (OrderInvoiceView *) ViewFactory::getView( anOrder );
    // print head of invoice
    invoice  << "Invoice to customer" << pInvoice->iCustomer.iCustName
             << endl;
    // ... and so on ...
    pInvoice->iSumOfInvoice = 0;
```

```
    // print invoice positions
    POSITION pos = pInvoice->iPositions.GetHeadPosition();

    while (pos != NULL) {
        OrderPosition &InvoicePosition =
            pInvoice->iPositions.GetNext( pos );
        invoice  << InvoicePosition.iArticle.iArticleNumber   << '\tab'
                 << InvoicePosition.iArticle.iArticleName     << '\tab'
                 << InvoicePosition.iQuantity                 << '\tab'
                 << ( InvoicePosition.iQuantity *
                      InvoicePosition.iArticle.iArticlePrice )
                 << endl;
        pInvoice->iSumOfInvoice +=
             ( InvoicePosition.iQuantity *
              InvoicePosition.iArticle.iArticlePrice );
    }
    // print sum
    invoice << "Invoice Sum :" << pInvoice->iSumOfInvoice << endl;
    // ... and so on ...
    // the view has been changed
    pInvoice->markModified();
}
```

Listing 1: printInvoice Implementation

For reasons of simplicity we did not show transaction brackets round the Order::printInvoice method. See the ⇒ *Transaction Object* pattern for this aspect. The method is static as the pattern is mostly used with data abstraction modules. 3GL languages will not allow object instances but are well suited to work with our framework.

Variants There are applications that define views only with regard to one table or a parent/child (1:n) relation. In these cases we may live without a full graph notation. The framework may be used unchanged but such simple projects will not yield the economic justification to implement the framework from scratch.

Some insurance companies use an enhanced form of logical data models that allow the specification of histories and time constraints. These companies are not only interested in a contract but are interested in the state of contract at a given time *t* [Sch96]. To allow the application kernel to navigate such data models, conditions and navigation edges must be enriched with expressions for time based navigation [Wür96]. The problem of histories is also present in an array of other application domains.

Known Uses VAA, an architecture for German insurance companies [VAA95], uses ⇒ *Hierarchical Views* together with time navigation. This is currently being specified in the so called Data Manager Component. Württembergische Versicherung [Würt96] is developing a so called data manager that uses ⇒ *Hierarchical Views* and offers a proprietary case tool to define them.

Many of sd&m's projects have used simpler forms (1:n views) of the Hierarchical Views pattern together with a view generation scheme.

Consequences The graphs defining hierarchical views may be defined using text files or even a specialized case tool [Würt96]. This allows automatic generation of the ConcreteView objects for statically typed languages or even runtime definition for dynamically typed languages. We have an interface that is expressed in terms of the logical data model. The application can easily navigate through the tree structured ConcreteViews without any knowledge of the underlying physical database.

See also A ⇒ *Query Broker* may be used to decouple ⇒ *Hierarchical Views* from the underlying physical database. The coupling to ⇒ *Physical Views* can also be implemented using hard wired dependencies. To avoid multiple database accesses for the same physical data we use the ⇒ *View Cache* pattern. To provide an interface to creation of views we use an ⇒ *Abstract Constructor* [Lan96] encapsulated in a ⇒ *View Factory*.

Pattern: Transaction Object

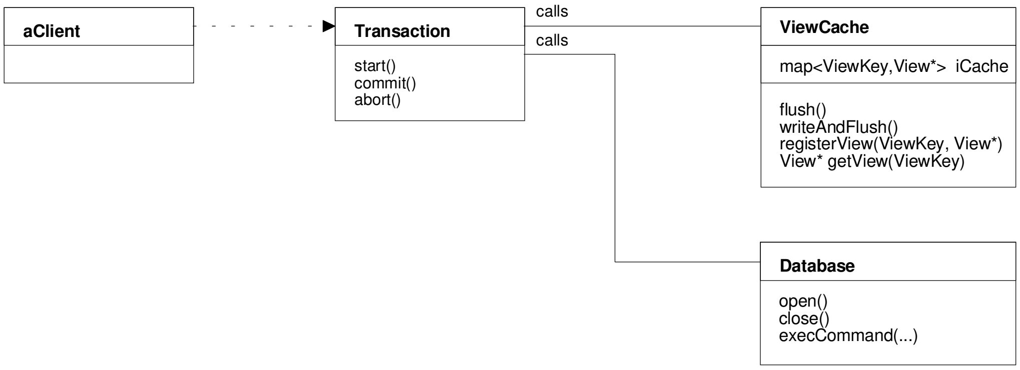
Context You applied the Relational Database Access Layer Framework. You need to define an interface for the start, commit and abort of transactions.

Forces An interface for transactions should separate transaction specific concerns from other concerns in the database access layer. The interface should be easy to use and easy to create. Collaboration with the exception handling mechanism is also necessary. Transactions must abort, whenever an unexpected exception arises and is not treated properly.

Problem What will the interface for transactions look like?

Solution Use a transaction object. Have it rollback the transaction in its destructor so that open transactions are aborted in any case if not committed explicitly.

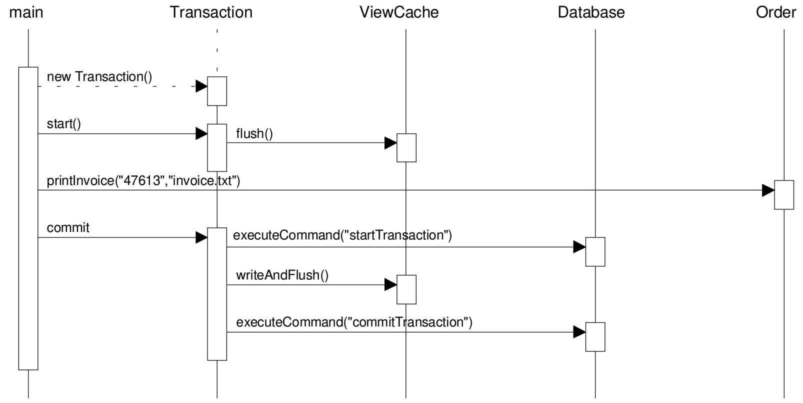
Structure The Transaction is responsible for flushing the ViewCache when a transaction is started. During the transaction the ViewCache is filled with View objects. When a transaction is committed, the Transaction object issues a writeAndFlush() command to the ViewCache.



Dynamic Behavior As an example we will show the lifecycle of a transaction object that is created by a main program. The locking strategy used is optimistic. The scenario is given by the following piece of user code.

```
void main ( void ) {
try {
    Transaction trans;
    trans.start();
        Order::printInvoice("47613","invoice.txt");
    trans.commit();
}
catch (...) {
    // trans is automaticaly destroyed here, resulting in an abort of the transaction
}
};
```

This results in the below interaction diagram.



Implementation The transaction object should be implemented as a ⇒ *Singleton* [GOF95] as relational databases will usually not allow you to use nested transactions. Use the ⇒ *Singleton* to prevent users from creating more than one (nested) transaction at a time.

You should also control the protocol of a transaction (no commit before a transaction is started) by using an internal state machine. The ⇒ *State Pattern* [GOF95, Dys+96] may be used to avoid conditional code in Transaction objects due to dealing with internal states of the transaction like *undefined*, *started,* or *aborted*.

Consider using the ⇒ *Strategy Pattern* [GOF95] if you want to give the API's user the option to choose a locking policy for each transaction.

Variants The interaction with the database is dependent on the locking policy used. In case of optimistic locking, the database transaction will be started just before the writeAndFlush() command is issued to the ViewCache. If the designer considers pessimistic locking more appropriate, the database transaction is started right after the ViewCache::flush() command resulting in a longer span of the transaction. Usually you will implement only one locking policy for an application. If you are building a reusable framework you should think of parameterizing the locking policy.

Known Uses The interface shown here can also be found in the ODMG C++ language binding for object databases [ODMG96].

Consequences The Transaction objects allows clean encapsulation of transaction behavior at the cost of a relatively simple class. The use of a class also ensures that instances are destroyed at the end of a block. Transactions cannot be left open by negligence or in case of an unexpected exception.

See also Brown and Whitenack describe a similar mechanism, called ⇒ *Transactions as Blocks*, for Smalltalk [Bro+96].

Pattern: View Cache

Context You write a sequence of methods that uses the same ConcreteViews over and over again in a single user transaction.

Forces Caching ConcreteViews can save you much time. Database access down to the physical disk level costs up to a factor of 100.000 more than an ordinary memory access. Even a well dimensioned database cache will usually not guarantee maximum performance. Databases will normally run in a process separated from user processes. To access the database cache you must access this other process.

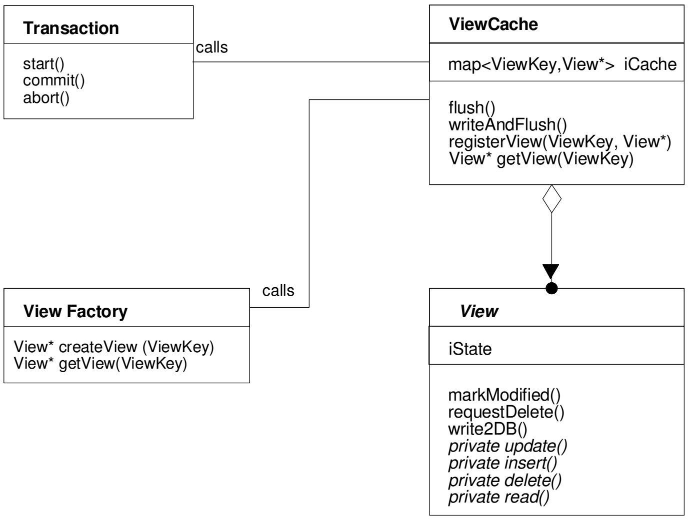
Process context switches however, are very expensive, even on a host computer. If the database process resides on a remote server, a local cache is even more important.

Another aspect is correctness. If you do not control multiple copies of ConcreteViews that may be requested by a coincidental sequence of application kernel methods, you run into lost update problems.

Problem How do you implement a cache for ConcreteViews?

Solution Create one cache of Views per database client process. Base it on a container that maps abstract database keys to pointers to Views.

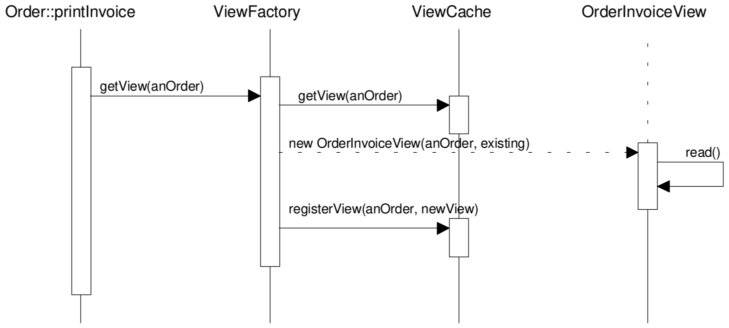
Structure The ViewCache offers usual cache methods that allow getting a View, registering a View in the cache and flushing the cache. You should separate any concerns of transaction handling from the ViewCache and leave them to the Transaction object. Aspects of how to write a View to the database should be placed in the ConcreteView objects. ConcreteViews are accessed via the abstract View protocol class.



Dynamic Behavior To illustrate the behavior of the ViewCache we will have a deeper look into what when executing the following code:

```
// get the appropriate data
OrderInvoiceView * pInvoice =
     (OrderInvoiceView *) ViewFactory::getView( anOrder );
```

The code is taken from Listing 1 above. Let's presume the OrderInvoiceView identified by the ViewKey anOrder has not yet been loaded. The ViewFactory will first try to ask the ViewCache for the View. As the ViewCache does not hold the required View, the ViewFactory will create a new View and have it loaded with data from the database that match the given ViewKey. The ViewFactory will then register the freshly loaded OrderInvoiceView with the ViewCache and return it.



Implementation The implementation of the ViewCache may be based on some Map container class.

```
class ViewCache {
public:
    static void flush ( void );
    static void writeAndFlush ( void );
    static void registerView ( ViewKey aKey, View *pView );
    static View *getView ( ViewKey aKey );
private:
    static Cmap<CString,LPCSTR,View *,View *> iCache;
};
```

It would be even more elegant to define a cache class template, instantiate that for Views and implement it as a singleton. The header given is sufficient to demonstrate the cache's idea.

Consequences A cache may cheaply be implemented using a hashed container like a map. The runtime penalty can be neglected compared to the penalty for a single superfluous database access. There is no memory penalty in case references or pointers are used instead of deep copies. Lost updates are prevented.

See also Brown and Whitenack describe a ⇒ *Cache Management* Pattern [Bro+96] in less detail. VAA calls their analogous mechanism a ⇒ *Session Memory* [VAA95]. The pointers returned by the getView() method can be made safer using ⇒ *Smart Pointers* [Mey96] or the ⇒ *Counted Pointers Idiom* [Cop92]

Pattern: View Factory

Also Known As ⇒ *Abstract Constructor* [Lan96]

Context You are using an object-oriented language for the implementation of the Relational Database Access Layer Framework. You want to implement a single factory for all objects derived from the View abstract class. This ViewFactory should work together with the ⇒ *View Cache*.

Forces The languages we are using for implementing the Database Access layer will usually not support RTTI. However the View base class should not need to be changed when a new ConcreteView class is added.

A View Factory also has to take into account that client code needs different exception behavior for Views that have to be activated from the database (read() followed by update()) and Views that do not yet exist in the database (insert()).

Problem How do you implement a single factory for all View subclasses that determines the right subtype from the given database key?

Solution Implement an ⇒ *Abstract Factory* [GOF95] using an ⇒ *Abstract Constructor* [Lan96] and shield it with an interface that distinguishes between new objects and objects that are activated from the database.

Structure The ViewFactory is responsible for creating new Views and for activating existing ones. It works together with the ViewCache as explained in the ⇒ *View Cache* pattern. The basic interface needed here is given by the following class declaration:

```
class ViewFactory {
public:
    static View * createView ( ViewKey aKey );
    static View * getView (ViewKey aKey);
};
```

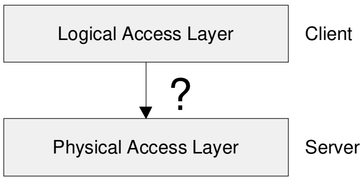
Implementation When implementing a View Factory you may use the following patterns. The ⇒ *Abstract Constructor* pattern [Lan96] contains a detailed description of how to implement a ViewFactory. Usually ⇒ *Prototypes* [GOF95] are used to trick statically typed languages. As stated above in the ⇒ *View Cache* pattern, pointers returned by the getView() method can be made safer using ⇒ *Smart Pointers* [Mey96] or the ⇒ *Counted Pointers Idiom* [Cop92].

See also The equivalent of the *View Factory* for non-object-oriented-languages is some other form of memory allocation for Views.

Pattern: Query Broker

Example Consider again the example we used to explain the ⇒ *Hierarchical Views* pattern. We define a Hierarchical View as a DAG in terms of the logical data model. We show how to *use* a Hierarchical View. But we did not show how to fill it with data from ⇒ *Physical Views*.

Context A central point in the Database Access Layer Framework is the question of how to couple the Logical Access Layer with the Physical Access Layer. Its an abstract formulation of the question *„How do we fill ConcreteViews from PhysicalViews and how do we write data to the database unloading ConcreteViews into PhysicalViews and further down to the database?“*



Forces The way components are coupled has a major impact on the cost, flexibility and reusability of a software system. The cheapest way on a short term basis is hard-wired coupling via function calls. A ConcreteView knows explicitly which PhysicalViews it has to call. The knowledge is hard-coded.

This works fine in case both layers are considered stable. The ⇒ *Layers* [Bus+96] pattern contains an example of layers that can be coupled using function calls. A layered stack of communication protocols is well suited for this coupling method as protocols are intended to be stable over years and not to change in short periods of time.

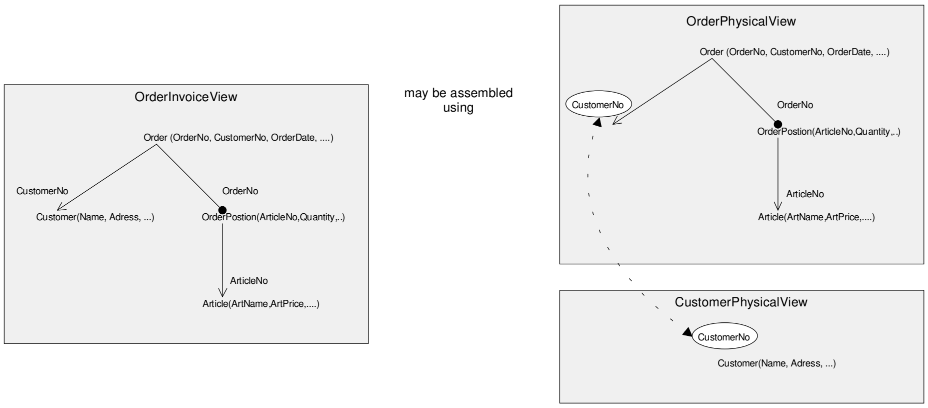
Whenever one layer is unstable and subject to frequent changes, some form of decoupling should be used. This observation can be made for fast changing user interface layers and stable application kernels. Software designers will decouple these by one side anonymization using ⇒ *MVC (Model-View-Controller)* [Bus+96] or ⇒ *PAC (Presentation-Abstraction-Controller)* [Bus+96] patterns.

Our database access problem is similar, except that the stable layer is on top of a rapidly changing Physical Database Access Layer. This calls for some form of decoupling. An alternative to using a decoupling mechanism would be using a big program generator that has to regenerate both layers whenever a PhysicalView is changed. This approach works fine for small systems and rare occurrence of changes. It will be expensive in terms of compilation and software distribution cost for large systems. It will produce deadly costs for large client/server systems with thousands of clients that need to be installed with megabytes of new database access layer software for each change in the physical database model.

Problem What kind of decoupling mechanism is best suited for the two layers in the Database Access Layer framework?

Solution Use a broker and call him *Query Broker*. The ConcreteViews are considered the client side of the broker. The PhysicalViews are considered the server side. The service requests are formulated as DAGs in term of the logical data model.

Structure Using a Broker is a standard method for decoupling. Usually symbolic, location transparent names are used to request a service. In our case, names are not appropriate. We need other ways to request services and register servers here. The left side of the Figure below shows the Hierarchical View description we have already introduced in the ⇒ *Hierarchical Views* pattern. The right side shows similar descriptions for two physical views. The upper one represents a PhysicalView on top of the Order and OrderPositionOverflow tables presented before in Figure 1. The lower one represents a physical view on top of a Customer table. Both can be linked together using the Customer’s primary key that is shown in a white ellipsis.



As we need the tree like descriptions anyway to be able to automatically generate ConcreteView classes (see ⇒ *Hierarchical Views*), we can also use them to describe the services offered by the PhysicalView classes. The QueryBroker is able to match the service request with services offered by registered PhysicalViews and delivers them as one QueryResult container.

The rest is a standard application of the ⇒ *Broker* pattern [Bus+96]. See the pattern for an explanation of the class structure and the dynamic aspects.

Example Resolved

Now we know how to fill the OrderInvoiceView in our example. The OrderInvoiceView passes the QueryBroker a request and the view description as a parameter. The QueryBroker checks his registered PhysicalViews for appropriate candidates, performs tree matching to assemble a query plan and calls the appropriate PhysicalViews. In our case these servers are the OrderPhysicalView and the CustomerPhysicalView. The QueryBroker then assembles the results and delivers them in a QueryResult container.

Implementation

When implementing the pattern you must deal with the following implementation issues:

- *Server Registration*: Especially in C++ you need a way to register all PhysicalView classes as servers with the QueryBroker before the program starts. This is typically done by misusing a static member that is initialized with some irrelevant contents and registers the class as a side effect.
- *QueryResults*: We did not treat the question of how to design the QueryResult containers. It is appropriate to use trees labeled with names from the problem space. The problem space is the logical data model.
- *Responsibility for Casting Data Types*: We did not yet assign the responsibility for casting data types from raw database types to application data types and vice versa. For example a CHAR(20) must be converted into an OrderKeyType and vice versa. We could leave that to the ConcreteViews as well as to the PhysicalViews. Both alternatives may be supported by a runtime dictionary that contains the logical data model and the knowledge which attributes to cast into which application data types.
- *Ambiguous Service Definitions:* We might run into situations where we have many server that deliver similar services. We can find several valid query plans for a given request. These might differ in speed. The Query Broker's tree matching algorithm has to be prepared to deal with ambiguous derivations. Similar problems can be found in optimizing compilers that deal with ambiguous grammars for code generation purposes [Kel91] and have to emit the cheapest code (find the cheapest query plan).
- *Query Descriptions*: The Client may register a query description once for a whole session or can as well send a query description with each request. The tradeoffs involved are similar to static versus dynamic SQL. If queries are used quite often it is cheaper to register them and store a query plan instead of starting the tree matching algorithm over and over again.

- *Form of Query Descriptions*: The Client and Servers of the Query Broker need a common form of describing Views (⇒ *Hierarchical* and ⇒ *Physical Views*) to enable tree matching by the ⇒ *Query Broker*.

Variants A further refined variant of a ⇒ *Query Broker* allows a very flexible development process. A ⇒ *Query Broker* that is also enabled to generate on-the-fly dynamic SQL queries can be used to substitute the Physical Access Layer during early development stages. Later, more and more Physical Views using static SQL may be added. If even this proves to be too slow, the Physical View implementations can be made even faster using the tuple interface of the database. The application kernel is shielded from such tuning by the ⇒ *Query Broker* who responds to request with the fastest services he can find in his repository of registered services.

Known Uses The *Query Broker* is a compilation of various best practices we know from our practice in the field of relational and object to relational access layers. Brown and Whitenack describe a ⇒ *Broker* [Bro+96] on a per class basis. This is like having one QueryBroker class for each ConcreteView class in our framework and not fully satisfying. The idea to define ConcreteViews and PhysicalViews in terms of a logical data model stems from the VAA Data Manager [VAA95, Würt96]. As the VAA Data Manager uses a generated hard wired coupling, this is not an appropriate scheme for client/server environments. Our experiences at HYPO-Bank [Kel+96, Col+96] taught us to use dynamic descriptions wherever possible. Two projects at sd&m used other important parts of the approach. The idea to start with dynamic SQL and migrate to static SQL or even to a tuple interface was born from bad experiences with a very slow database server in a project, we may call X. As project X was conducted using a prototyping approach, the project makes extensive use of a runtime data dictionary and allows to use fully dynamic queries or precompiled queries alternatively. Another project Y uses tree matching for query by example style combination of query plans on a large data model that is subject to very rapid changes.

Consequences *Cost versus Flexibility:* The Query Broker offers two side anonymization for client/server communication between the Logical and Physical Access Layers of our framework. A Query Broker is expensive to implement for a single project due to the need for tree structured result containers, request descriptions and a tree matching algorithm. A run time dictionary that contains the logical data model and supports casting and view generation is another extra that increases the cost of such a solution. Implemented in a reusable framework and used in far more than one application this will pay off against the cost of dealing with less flexible, hard wired code generation solutions that make optimizations more expensive and dealing with some thousand client sites a nightmare due to reasons of software distribution.

Generated Access Layer Modules: Many projects rely heavily on generation of access layer modules and hard coded mappings. It is possible to generate access layer modules, consisting of several embedded database language statements plus error handling from compact table descriptions. This is achieved using macro languages like Perl [Wal+90]. This is a very reasonable approach, as long as only static SQL or some other static embedded data manipulation language is used. With dynamic SQL or another interpreted data manipulation language it is better to use a ⇒ *Query Broker*. The Rolls Royce is, to build an access layer that allows use of static and dynamic SQL in parallel. The ⇒ *Query Broker* is a means to provide such functionality. This offers the possibility to use dynamic SQL during a development phase and static SQL or even a lower level database API for production. The static database interfaces should be generated.

See also The pattern is an application of the ⇒ *Broker* [Bus+96] pattern in the context of our framework.

Pattern: Physical View

Context In the process of optimizing a physical database scheme for high speed access, you will use techniques like ⇒ *Denormalization*, ⇒ *Overflow Tables*, or ⇒ *Controlled Redundancy*. In Figure 1 we have created two tables that belong together and deliver a view that contains a repetition group of OrderPositions belonging to a single Order.

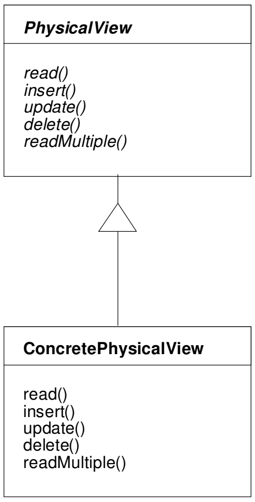
Forces Now imagine your boss tells you to implement a ⇒ *Query Broker* using dynamic SQL only on top of such a table layout. You might spent some time to do that straight or you might better try to factor out complexity that was brought into your application by table optimizations. So the tradeoff here is complexity of code using the database versus introducing yet another layer of software. The optimization patterns mentioned above are pretty schematic. So we might hope that the code to handle them might also be schematic. This calls for code generation out of some description to make creation of the extra layer of software easier.

Another requirement is use of static SQL or even lower level database APIs. Hence we need some place to implement static queries. The ⇒ *Hierarchical Views* patterns brought us to the conclusion that SQL-code related to the physical database scheme should be handled in a lower layer. Such unstable code should not mess up the stable layer of the application kernel's ⇒ *Hierarchical Views* interface. The last requirement here is that the database has to be encapsulated as a server for the ⇒ *Query Broker*.

Problem How do you hide the messy handling of table optimizations and how do you create an elegant to use interface to your physical database?

Solution Encapsulate tables that are related by optimizations into so called ⇒ *Physical Views*. Create an interface that can be described in terms of the trees describing ⇒ *Hierarchical Views*. Derive ConcretePhysicalViews from a common protocol class that eases handling by callers.

Structure PhysicalViews are responsible for providing an interface on the subset of data they are defined for. PhysicalViews may cover very different areas of the database. In the above example for the ⇒ *Query Broker* pattern we have defined the OrderPhysicalView that offers parts from three logical entities optimized into one table. This view should typically not allow an update of Article data (due to ⇒ *Controlled Redundancy*).



On the other hand the same example contains a simple view on one entity, the Customer, that is represented using one table. You might also create ⇒ *Narrow Views* for filling list boxes while at the same time offering a full size view for editing an entity. This will sure happen to the Customer entity. In practice there will also be views that cover one entity that was split into several tables due to optimization reasons.

Implementation When implementing the pattern you will mostly be confronted with implementation issues already discussed in the context of the ⇒ *Query Broker* as we did not further define the ⇒ *Query Brokers* interface to ⇒ *Physical Views*.

A generator for ⇒ *Physical Views* is highly recommended.

We could not find a cookbook for the definition of ConcretePhysicalViews. A reasonable first cut approach is to provide views for each independent entity like Customer or Article and to define combined (1:n) views for parent/child relations like the Order/OrderPosition relation. Further ConcretePhysicalViews have to be provided to hide physical table scheme optimizations like the above mentioned ones. Maybe there's an own pattern language for finding views. If so we did not yet investigate it.

Variants PhysicalViews may also be used to encapsulate non-SQL databases and file handling mechanisms like ADABAS, IMS-DB, CODASYL databases and VSAM files. As mentioned in the Relational Database Access Layer pattern, this variant may be used to build relational applications on top of such databases.

Consequences PhysicalViews induce an extra layer of software in the Relational Database Access Layer Framework. They may be skipped as long as a project runs in a prototyping phase with the physical data model being a non-optimized copy of the logical data model.

If ⇒ *Chapter 5's* optimization patterns are used, this calls for adding the extra layer to hide the complexity of otherwise hard to handle optimized table schemes.

See also The optimization patterns described in ⇒ *Chapter 5* are closely related to the creation of physical views.

5 Some Database Optimization Patterns

This chapter introduces a far from complete collection of database optimization patterns for use with relational databases. The Database Access Layer Framework is designed to give the developer the freedom to use these patterns without affecting the application kernel code. Database optimization techniques may be applied to the physical database scheme whenever the actual workload of the database requires performance improvement. This will happen in recurring intervals as changing contents and changing use of a database will require adaptation of the physical database scheme.

Pattern: 80/20 Optimization Rule

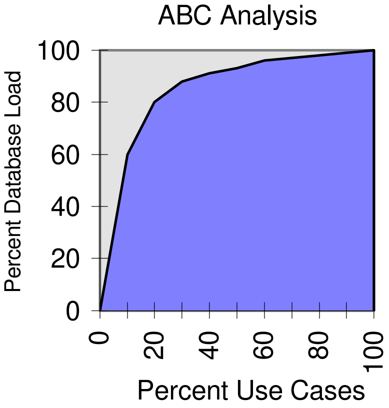
Also Known As *Pareto Principle* or *ABC Analysis*

Context Users tell you the response times of the relational database based system you built for them are far from acceptable. After checking you find out that you do face a performance problem with dialog reaction times above three seconds. You know you are facing a database performance problem.

Forces Whenever you run into performance trouble with an information system that is predominantly database driven, the most likely place where you will find the reasons for that trouble is the database. Checking the code and optimizing machine instructions will seldom yield significant results as database accesses count in hundreds of milliseconds while execution of code counts in nanoseconds. Database optimization however is time consuming and expensive. Database optimization requires test runs with real data and real usage conditions. People therefore often hate to attack database related problems.

Problem Where do you start optimizing the physical database scheme?

Solution Perform an ABC analysis and find the 20% or less of all ⇒ *Hierarchical Views* that cause 80% or more of the databases load. Optimize exactly these using the patterns provided in this chapter. Try to handle as many of these use cases as possible with one single indexed key access. This access should best touch only a single physical page of the database.



Consequences Having optimized the critical use cases you will see an overall performance improvement. You will not see something like optimal performance for all use cases. Improvements to the most important use cases might even result in performance degradation for less important use cases.

If you concentrate on important use cases you will be able to show fast results at acceptable cost.

See also The most likely improvements to start with are ⇒ *Short Views*, ⇒ *Denormalization*, ⇒ *Controlled Redundancy* and ⇒ *Narrow Views*.

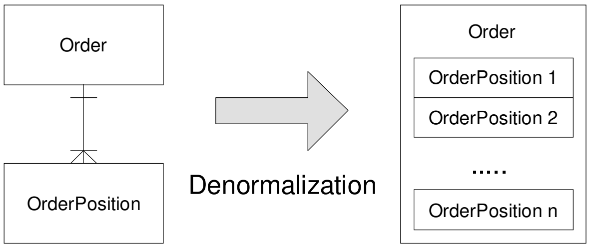
Pattern: Denormalization

Context The analysis you have started following the ⇒ *80/20 Optimization Rule* shows that an Order View is one of the most heavily used views in the database. Reading an Order and the OrderPositions costs a join statement and several physical page accesses. Writing it costs several update statements. Further statistical analysis, again using the ⇒ *80/20 Optimization Rule,* yields that 90 percent of Orders have no more than 5 positions.

Forces Database optimization is mostly a question of time versus space tradeoffs. Normalized logical data models are optimized for minimum redundancy and avoidance of update anomalies. They are not optimized for minimum access time. Time does not play a role in the denormalization process. A 3NF or higher normalized data model can be accessed with minimum complex code if the domain reflects the relational calculus and the logical data model based on it. Normalized data models are usually better to understand than data models that reflect considerations of physical optimizations.

Problem How can you manage to read and write most physical views with a single page database access when you have a parent/child (Order/OrderPosition) relation?

Solution Fill up the Order entity database page with OrderPositions until you reach the next physical page limit. Physical pages are usually chunks of 2 or 4K depending on the database implementation. The number of OrderPositions should be near or greater than the number of OrderPositions that reflects the order size in 80-95% of cases.



Consequences You need more disk space in the database depending on alignment rules. If you are lucky and the database system will begin a new page with every new order the space is even free. As you have to deal with the 5-20% or so cases of orders that do not fit into the Order/OrderPosition record, the code will become more complex. The database will also become less queryable for ad hoc queries as the physical structure does no longer exactly reflect the logical data model.

See also Use an ⇒ *Overflow Table* to deal with the other 5-20% of orders that have a mandatory number of OrderPositions. Use ⇒ *Physical Views* to Encapsulate the code that is needed to handle the cases of page overflow.

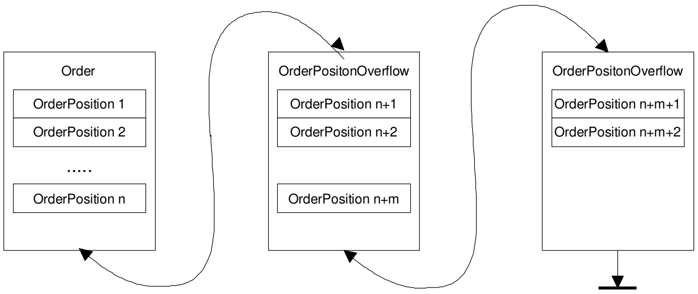
Pattern: Overflow Table

Context You have followed the ⇒ *Denormalization* pattern's advice and have denormalized a relation.

Forces Correctness forces you to find a solution that allows you to access positions that do not fit on the Order page. A possible solution should not mess up the client code and should also be designed for good performance. The time/space tradeoff cited above is present here as well.

Problem What do you do with those orders that have more positions than the number that you did integrate into the Order table?

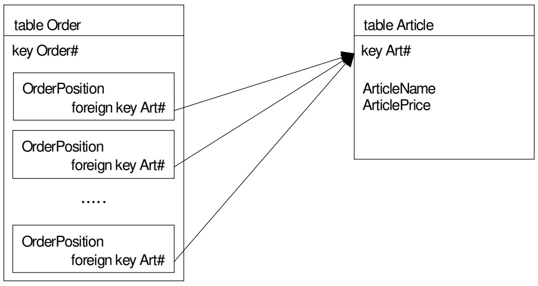
Solution Use a second table, a so called overflow table, that contains another physical database page full of OrderPositions. Identify the OrderPositionOverflow table with a dumb number and link it to following or previous OrderPositionOverflow records by fields that contain the predecessor and the following record that might contain more OrderPositions.



Consequences You have solved you correctness problem at the cost of more complex code. You have achieved maximum possible speed for those orders that fit into Order/OrderPosition table. The space consumption is less than optimal for such orders that use only a small percentage of an *Overflow Table's* record.

See also You should use the ⇒ *Physical Views* pattern to encapsulate the messy conditional code that arises from handling the double linked list of OrderPositionOverflow records. Denert describes Denormalization as a case study on physical database design in [Den91, pp. 285-288].

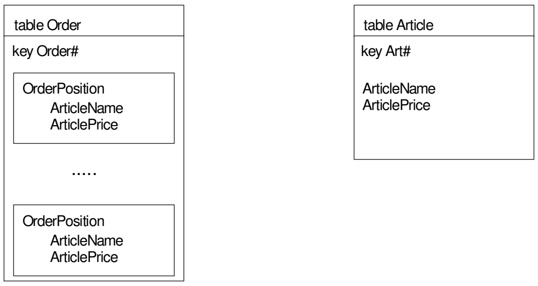
Pattern: Controlled Redundancy



Context In the process of optimizing the order processing system you find out that clerks often produce a test invoice and check it at the screen before they confirm orders. They complain to you that this use case could be handled faster. From database trace protocols you find out that the queries for printing orders produce heavy database load. You did denormalize the Order table. But for a normal invoice with five positions you still have 7 databases accesses. One for the order and its positions, one for the customers data and 5 primary key accesses for the article data needed for five order positions. You consider this unacceptable.

Forces The forces involved here are the same as in the ⇒ *Denormalization* pattern.

Problem How can you manage to read physical views with a single page database access when you need to read data from a parent entity?



Solution Replicate those parts of the parent entity (Article) in the child entity (OrderPosition) that you need for the use case. Replicate only stable data that are not subject to frequent updates.

Consequences You can considerably reduce the number of database pages accessed for read operations by using ⇒ *Controlled Redundancy*. In case of write operations to the parent entity (Article) you will need additional database accesses as you have to update more than one table to write the same (redundant) fact. ⇒ *Controlled Redundancy* should only be used for stable data. An article name will seldom change during the live span of an order. Redundant customer data like the customer's name will also seldom change. It is acceptable to replicate such data. If it comes to an article price we may well start a discussion. If the price is fixed during the live span of the order, you will replicate it. If the price is subject to frequent updates you should consider the read / update performance tradeoff.

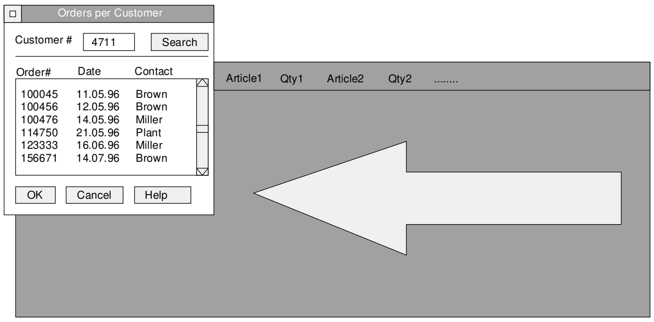
You also need more database space for redundant data. *Controlled Redundancy* results in more complex code as you pull application level considerations down to the database level. A database user must know that he or she also has to update the Order table if the name of an article changes. In any case *Controlled Redundancy* should be shielded and controlled by a Physical Access Layer. Never clutter code that accesses such data into the application kernel.

Redundancy does not impair the understandability of a physical data model. It might even improve understandability.

See also ⇒ *Narrow Views* will motivate you to replicate only the data you really need for high performance use cases.

Pattern: Narrow Views

Context For an order processing system you want to design a dialog box to select orders by customer and to view them. You have already defined a ⇒ *Physical View* for orders and as you have use cases that process more than one order at a time, you have implemented a readMultiple() method that delivers a set of orders for a given customer.



Customer # 4711

Order#	Date	Contact
100045	11.05.96	Brown
100456	12.05.96	Brown
100476	14.05.96	Miller
114750	21.05.96	Plant
123333	16.06.96	Miller
156671	14.07.96	Brown

Forces The natural way to fill a list box is a dynamic SQL query. Such queries are most flexible and are the best fit for query by example. Anyway some database administrators will not allow you to use dynamic SQL for reasons of access protection or database load as user defined queries cannot be controlled by a central database administration. They can cross police lines in an unprotected database and might also cause uncontrollable database load in large online systems, impairing other users. So the views you need will typically be shielded by ⇒ *Physical Views.* Each physical view is a code object. An explosion of code objects is always considered negative. On the other hand you always want to economize on bandwidth. You want to transport only those data from the database that are really needed. This is especially important if you have a Client/Server cut between the object that consumes the data (the list box) and the object that provides the data (the database).

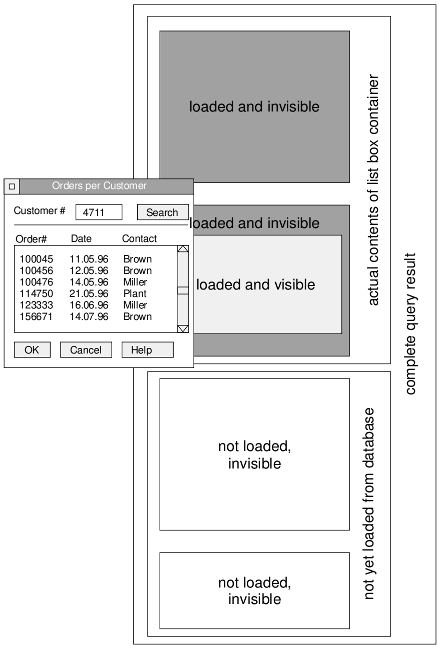
Problem What kind of views should you use for filling list boxes?

Solution Use narrow views . 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.

Consequences You will have a second set of ⇒ *Physical Views* for filling list boxes. This increases the amount of code objects in a system. On the other hand you have maximum efficiency.

See also For reasons of clarity we have factored out the second rule for list boxes in the ⇒ *Short Views* pattern.

Pattern: Short Views



Customer # 4711

Order#	Date	Contact
100045	11.05.96	Brown
100456	12.05.96	Brown
100476	14.05.96	Miller
114750	21.05.96	Plant
123333	16.06.96	Miller
156671	14.07.96	Brown

Context Given the list box problem in the ⇒ *Narrow Views* pattern you have designed a ⇒ *Physical View* that contains exactly the data for filling the list box. Users still complain that filling a list box can take up to 20 seconds for the best customers. Good customers may have hundreds of open orders associated to them.

Forces The query that is used to fill a list box will often yield far more records than there are lines in a dialog box. Many list box classes are therefore based on a container that can handle an arbitrary number of list box items. Fetching a large quantity of records at once will have two implications: Often you might only want to see a few pages of information. In such a case a dumb query causes enormous database load. You are also blocking bandwidth while the list box is filled with lines you cannot see and worst - the user is gazing at a hourglass pointer.

Problem How do you speed up filling of list boxes and how do you prevent unnecessary data from being loaded into the list box?

Solution Load data in chunks that allow a reasonable response time. A rule of thumb is 30-50 records for C/S system. This is equivalent to two times the number of lines in a list box.

Variants There are two possibilities to implement the fault mechanism when the user has come to the end of the sequence of already loaded objects. If the programming environment supports database cursors (see *Database Cursor for Selects [Sie96]*) the list box container will simply ask the cursor for more records. If you are working with a transaction monitor that does not allow you to have open cursors over dialog steps you have to issue a new query starting from the last record you have in a listbox and yielding no more records than the fixed number you want each time you access the database.

Consequences Using this scheme results in better response times for the user and less load for the database. Some more code is needed to implement a container that asks the database for more records. This code can well be encapsulated in a smart container behind the list box class.

See also The easier variant to handle block reading is covered by the ⇒ *Database Cursor for Selects Pattern [Sie96]*. The ⇒ *Result Streams Pattern [Bro+96]* covers containers that send a request for more data if the user reaches the end of the current set.

6 References

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

[Cop92] **James O. Coplien:** *Advanced C++, Programming Styles and Idioms*, Addison Wesley 1992.

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

[Den91] **Ernst Denert**: *Software-Engineering*, Springer Verlag 1991.

[Dys+96] **Paul Dyson, Bruce Anderson:** *State Patterns*, Preliminary Conference Proceedings EuroPLoP, First European Conference on Pattern Languages of Programming, Irrsee, Germany, 1996

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

[Lan96] **Manfred Lange:** *Abstract Constructor,* Preliminary Conference Proceedings EuroPLoP, First European Conference on Pattern Languages of Programming, Irrsee, Germany, 1996.

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

[Sie96] **Siemens AG, Unknown Author:** *DB Cursor for Selects*, Preliminary Conference Proceedings EuroPLoP, First European Conference on Pattern Languages of Programming, Irrsee, Germany, 1996

[VAA95] **GDV:** *VAA - Die Versicherungs-Anwendungs-Architektur*, 1. Auflage, GDV, Bonn 1995

[Wad94] **Drew Wade:** *Options for persistent Objects*, Proceedings ObjectWorld 94, London 1994.

[Würt96] **Württembergische Versicherung:** *Projekt Datenmanager***,** private communications 1995 - 1996.

[Wit96] **Andreas Wittkowski**; *Datenbankdesign & Performance*, slide presentation, sd&m Internal Lecture Series, 1996.

[Wal+90] **Larry Wall, Randal L. Schwartz**: *Programming in Perl*, O' Reilly 1990.