Object-Oriented Data Integration

Running Several Generations of Database Technology in Parallel

Wolfgang Keller, Christian Mitterbauer, Klaus Wagner.

Abstract

Large IS shops often work with three or more generations of database technology. It is very common to find some pilot projects, using object technology while the mass of software is still being produced using 3GL technologies and relational databases. Parallel to this application portfolio, older applications have to be maintained. These applications often use hierarchical or even older database technology. In most cases hierarchical databases still manage the main workload of commercial data processing. It is therefore important to be able to federate all the above generations of database technology.

Object-oriented application development has to be able to reach data provided by other software generations. This may not result in changes to older applications. Object-oriented databases as a technology are not sufficient to provide this kind of parallel data integration. This article introduces an integration framework for several generations of database technology. The framework can be filled with multiple categories of database, middleware and other products. These are introduced and discussed in a separate section.

As an example we will discuss a solution of the data integration problem for a large German financial institution, using an object-oriented access layer for heterogeneous databases. The focus here was on integration of all existing data sources. Object-oriented client/server applications have to coexist with classical host environments. This article features special conditions and requirements that can be found in a large IS shop. These requirements and the characteristic software environments that can be found there still rule out most commercially available integration products such as access layer products or object-oriented databases with relational gateways.

About the authors

Dipl.-Inform. Wolfgang Keller is a consultant with sd&m, software design & management GmbH & Co. KG in Munich/Germany. Dipl.-Inform. (BA) Christian Mitterbauer works for Bayerische Hypotheken- und Wechselbank AG also in Munich/Germany. Dr. Klaus Wagner is also a consultant with sd&m. Authors can be reached via the Internet: *Wolfgang.Keller@sdm.de*, cmitterbauer@*mail.hypo.de* and *Klaus.Wagner@sdm.de*.

This work has been partially sponsored by the German Ministry of Research and Technology under contract 01 IS 508 A 0 . We would like to express our thanks. Special thanks to Dr. Thomas Becker for proof-reading. You must have had a horrible time Thomas.

Contents

1 Introduction 1

2 Integration Frame 3

2.1 Programmers View 3

2.2 Layered Model 5

2.3 Object Layer 6

2.3.1 Functional Units 6

2.3.2 Discussion 7

2.4 Tuple Interface 8

2.5 Restrictions of an Integration Model 8

2.6 Further Reading 9

3 Product Categories 9

3.1 Remote Database Access Products 10

3.2 Objectified Relational Databases 10

3.3 Federated Databases 10

3.4 Object-Oriented Databases with Relational Gateway 10

3.5 Object-Oriented Access Layers 11

3.6 Object/Relational Databases 11

4 Case Study - Persistent Objects in a Large Bank 12

4.1 Situation, Requirements and Software Environment 12

4.2 Why Products Still Fail 13

4.3 Our Architecture 14

5 Experiences and Summary 16

6 References 17

1 Introduction

Many large IS organizations, like e.g. banks, are facing an introduction decision for object technology. Some or most have collected first experiences with object technology and have started pilot projects. This results in a coexistence problem of old and new technology generations. Big bang replacement strategies are seldom ever used and are not advisable for large IS shops [Bro+95]. Most likely, migration will take 10 or more years to complete. Several generations of software technology will have to coexist during such a time span. Old data resources will probably live on even longer than that.

Host applications will have to coexist with object-oriented client/server applications. Decoupling the two branches of development is essential. It is not realistic to expect that that there will be a clean separation of data resources for different generations of software development technology. Enterprise data models have been invented to provide tight integration of information systems over an organization and not to provide separate islands of data. They are still an enormous source of benefit for IS organizations. Nobody believes that this integration will be thrown away just because some new technology crops around. Technological renovation has to go along with solid business advantages.

The above factors result in larger IS shops being confronted with three to four[1] generations of database technology that have to be tightly coupled.

- We will use the term *first generation* for hierarchical database systems like e.g. IMS-DB. Hierarchical databases often still manage the main workload of commercial data processing.

- The second generation are relational databases like e.g. DB2. Most larger IS shops use at least generations one and two. The second generation is often still in the process of replacing the first one.

- Object-oriented database management systems (OODBMS) [ODMG93, Kim95, Cat94] can be seen as generation three. Products that can be found here are at the brink of use for mission critical applications. Some have well passed that border. Most IS shops still use them only in pilot projects if at all.

---

[1] Most large IS shops also use VSAM ore similar file systems instead of database systems to a signifficant extent. This could be called the zero generation of database systems - resulting in four generations of database technology that have to integrated.

The developer, regardless of whether the language is C++, Smalltalk, COBOL or PL/I, should be confronted with only that image of the enterprise data that conforms to his or her technology generation. Advantages of enterprise data models should not be thrown away. The same data resources must be useable by conventional host applications and object-oriented applications at a time. The data resources must be readable and *writeable* by all other technology generations. *„Writeable"* especially is the prime killer criterion for many integration products. Solutions are highly non-trivial.

The market for object-oriented database technologies is still small and unstable compared to the market for relational databases. Six months can be seen as a normal innovation cycle. This explains why many experts have given up writing books on the topic - they tend to be outdated the day they appear in a bookstore. IS managers can be as good as sure, their solution is technically outdated the very day it is implemented.

That is why we plead for a clean separation of concerns in any solution for an object data integration problem. This will allow projects to react to market movements by installing cheaper or better components than the ones selected for a first solution at project startup time.

There is no such thing as only *one* solution for the object-oriented integration of database technologies. But it is nevertheless possible to provide an integration model that is valid for many solutions in many different software environments and for many different constellations of specific requirements. Such a frame is then filled up with available components from the market that are integrated with self-written glue. The final choice of components for a specific project is dependent on the chosen project and the enterprise's own installed software environment consisting of database systems, programming languages, transaction monitors and middleware components. A limit for possible project costs and investments will usually be set by protential positive financial impacts the project is expected to have on business. The choice of products will also depend upon the actual object-oriented databases and integration software on the market at project start time.

Section 2 will introduce a proven integration model for data resources of various ages. Product surveys would be outdated the day they are written. This is why section 3 will list product categories instead of actual products. We will show how these categories can be integrated in our solution framework. Section 4 contains an application case study. The architectural framework has been used to create persistent client objects in a typical financial institution's software environment. The model has been extracted from project experience and not the other way round. This article summarizes some 5 years of experience gained in various data integration projects in our companies practice.

2 Integration Frame

The object data integration model describes a software architecture that provides persistency[2] for objects. This also allows to provide persistency for object-oriented applications on top of legacy data resources. The architecture can best be compared with an architecture for a very simple[3] object-oriented database system. The salient feature is the possibility to store data in arbitrary legacy data stores such as relational or hierarchical database systems. This provides an object-oriented integration mechanism for several generations of database technology. An integration of distributed data can also be supported.

One of the very slick features of object-oriented database systems is to give a programmer the impression of having to deal with objects only. Any database details are hidden as much as possible. An object-oriented database system will first provide a persistent programming language together with typical database features such as independence of data from a programming language, transactions, locking and more.

We will therefore first show a programmer's view of our integration model. After that we will discuss the internal structure of the integration frame.

2.1 Programmers View

The programmer's view is illustrated by an example of a typical interface for persistent objects in C++. This interface is as similar as affordable to the object-oriented database standard [ODMG93]. The example does by far not contain all possible constructs of the ODMG specification. A relatively small subset has proven sufficient for real world projects. Expensive constructs[4] such as OQL [ODMG93] have not been implemented. Our example is in no way complete. The user code example is there to present the basic ideas of a persistent programming language.

---

[2] Persistency is the property of objects to survive termination of a process and to be alive in the next process started, if required to do so [Atk+83, Sou94].

[3] The degree of reduction of functionality compared to an OODBMS depends on the possible investment in an object layer. In case of an OODBMS with a relational gateway that fits into the given environment, the functional properties are not worse than those of any OODBMS except maybe performance. In case of a custom solution it would be too expensive to implement advanced features like e.g. OQL, schema evolution, nested transactions, lanuguage mappings and such that should be part of an OODBMS.

[4] Expensive features to implement are e.g. OQL, schema evolution, nested transactions, lanuguage mappings.

```
void example ( String CustomerName ) {

try                             // database errors are handled by C++ exceptions


        Transaction trans;                      // create a transaction and
        trans.begin()                           // start it

                // Declaration of some variables for our example
                Ref<Customer> oldCustomer;      // Ref is a smart pointer to a persistent
                                // object
                Set<Ref<Order>> orders;         // Set<Ref<Order>> represents a set of
                                // persistent objects

                // Fetch a Customer
                oldCustomer = Customer::getByName(CustomerName);    // This will only load
                                // and check an ObjectId and assign a smart pointer. It's
                                // a database method.

                // Dereferencing a set of orders
                Orders = oldCustomer->confirmedOrders;      // The Orders variable
                                // is assigned a whole set of confirmedOrders.

                // Create a new customer
                Ref<Customer> newCustomer = new(Customer);

                newCustomer->Name = someValue           // the newCustomer object is changed
                                // by assignment of a value to an instance variable.

                // Assignment of whole set of orders
                newCustomer->confirmedOrders = oldCustomer-> confirmedOrders;
                newCustomer->markModified();            // marking object dirty
                                // results in object to be written to the database
                                // at time of next commit

                // Delete oldCustomer
                oldCustomer->requestDelete();

                // Finish transaction and commit it
        trans.commit()
}

catch ... // Error handling has to take place - but is not shown here
};
```

Figure 1: Sample User Code

The following actions are presented in Figure 1:

- Getting an *oldCustomer* by name from the database
- Dereferencing the customers *confirmedOrders*
- Creation of a new *Customer* instance by assigning the freshly created object to a smart pointer.
- Assigning a whole set of orders at a time.
- Deleting a *Customer* object by requesting its deletion.

One thing is invisible here - relational database code or actions. The application programmer's view of persistent objects is very similar to his or her view of volatile objects. The visible difference is methods like `getByName`, `markModified` or `requestDelete` and smart pointers `(Ref<SomeType>)`. The methods are acquired by protocol inheritance from an abstract base class *PersistentObject*. This

interface is not the ideal vision of a persistent object's interface where persistent objects cannot be discriminated from volatile objects. For a further discussion of the above interface see [ODMG93].

2.2 Layered Model

The apparatus necessary for implementing the above interface is comparable to an object-oriented database system with heavily reduced functionality. In the following the term *object layer* will be used for a system providing an interface like the above.

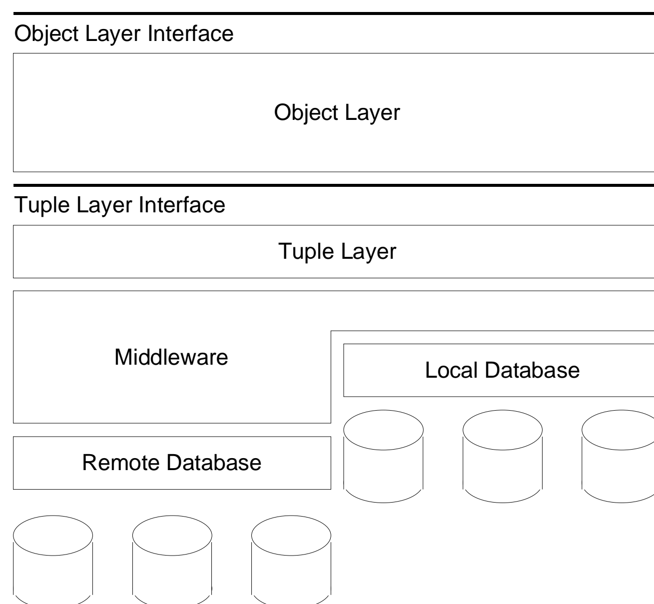

Figure 2: Persistent Objects Integration Model

Data is not stored in an OODBMS's data management system but in legacy database systems like RDBMS or hierarchical database systems. Therefore we need an interface that presents data from more than one database in relational tuple form secured by transactions. In the following this will be called the *Tuple Layer*.

The corresponding layered architecture is shown in Figure 2. The object layer's services are described in more detail in section 2.3. Section 2.4 lists requirements that have to be fulfilled by a tuple layer. The above model serves as an integration frame or architecture for a custom solution as well as for product solutions. The above frame can be filled with prefabricated or self-written components. Section 3 describes product categories for prefabricated components. These components span very different parts of the integration frame.

2.3 Object Layer

The object layer could be implemented using an OODBMS, a prefabricated framework or a custom solution. Each possible solution should be close to the [ODMG93] standard. The functional units are always similar. Figure 3 provides an overview of the important functional units in an object-oriented access layer.

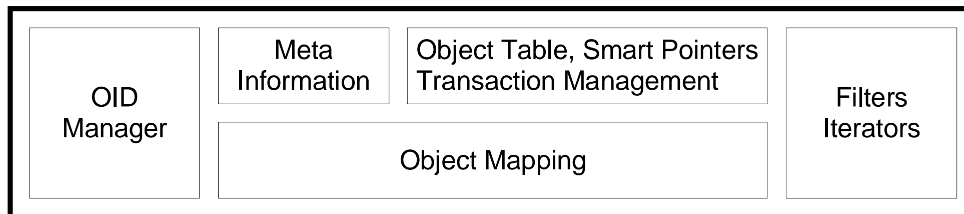

Figure 3: Functional Units in an Object-Oriented Access Layer

2.3.1 Functional Units

If legacy data sources are to be integrated, an object layer has to use some kind of tuple layer as its storage medium. This is why an integration architecture differs in its architecture from OODBMS products. The tuple layer offers an interface to relational tuples over primitive data types. Abstractions like e.g. complex data types, inheritance and relations are unknown at this level. They have to be implemented by the object layer.

Hence the object layer has to provide the following services to map full-fledged objects to a tuple layer:

- Complex data types and objects must be assembled from primitive data types. This involves casting raw data types into application data types as part of the *Object Mapping*.
- Mapping tuple fields to object attributes must happen with respect to inheritance and inter-object relations. This is also done by the *Object Mapping* with the help of *Meta Information*.
- Object identities have to be constructed from key fields at the tuple layer's level. The *OID Manager* takes care of this part.
- Associations at object level are built using foreign key fields and relation tables at the tuple layer level. This is done by the *Object Mapping* in collaboration with the *OID Manager*.

A description of all internal details of an object layer is beyond the scope of this article. There are several articles[5] that deal with special aspects of building an access layer in more detail.

Besides the services described above, further services have to be provided by an object-oriented access layer:

- There should be objects for *Transaction Management*.
- The identity of objects in volatile memory must be guaranteed by an *Object Table* [Hah+95].
- A *Smart Pointer* [Str91] mechanism is needed to prevent large chunks of objects from being loaded into volatile memory if a first object is touched that has transitive associations with a great number of other objects [Kel95b].
- There should be efficient support for list boxes. List boxes are typically filled with a selection of a few attributes from a very large set of objects. With respect to low bandwidth of today's client/server communications lines in wide area networks, it does not make sense to load complete objects for presentation in list boxes. Instead only the portion needed out of all possible attributes should be loaded. Blocked read operations should deliver only as much records as can be seen in a list box at a single time. *Filters* are used as a surrogate for OQL queries. Smart and lazy *Iterators* are used to browse the result sets of those queries.

2.3.2 Discussion

Object-oriented database products are sufficient in a pure object-oriented target environment. Such products may be evaluated using the object-oriented database standard [ODMG93] as a reference model.

If relational or hierarchical data sources have to be used and if data resources must be used in parallel with existing legacy applications, shrink-wrapped products can hardly be found. Detailed knowledge of interfaces between an object layer and storage management facilities is needed to evaluate products in this category. The cut between object layer and storage mechanisms might run right through a product. If a storage manager is to be substituted or written in such a situation, deeper knowledge of object-oriented versus relational storage concepts is needed.

---

[5] The different building blocks are described in more detail in an array of papers by different authors. See e.g. [Col+95, Hah+95, Kel95a, Kel95b, Kel+95, Lip95] if you are interested in more details.

Different product categories and their role in our integration architecture are discussed in Section 3.

2.4 Tuple Interface

A *Tuple Layer* [Col+95] is able to deliver a unified relational view upon relational and hierarchical data sources. Some user transaction construct is also provided. The task is comparable to a subset of a federated database system[Kim95]. A tuple layer should be able to cope with the following requirements:

- At least read, insert, update, delete and read multiple operations should be supported for each tuple presented by the *Tuple Layer*. These operations are identical to a simple relational access layer that is used in many non-object-oriented projects.
- The physical source of data is hidden by the *Tuple Layer*. The *Tuple Layer* provides a view of a single integrated database.
- The *Tuple Layer* is able to map hierarchical data resources to relational tuples.
- The *Tuple Layer* is able to provide transparent user transactions over multiple database systems. This is not a trivial task if data sources from more than one server are involved. In general a 2-phase-commit would be needed for this task. It is also not a trivial task if transaction servers like e.g. CICS or IMS-TM hosts are being used.

There are several options for implementing a *Tuple Layer*. The choices include federated database systems [Kim95], DRDA products [Orf+94] and remote-SQL-interfaces for less challenging requirements. The term *Middleware* in Figure 2 is used to sum up these choices. Depending on the power of products more or less attention for this middleware block is needed. Available product categories are also discussed in Section 3.

2.5 Restrictions of an Integration Model

Using the above integration model for mass updates[6] does not make too much sense. Executing mass updates at an object layer level could mean replacing efficient mechanisms of underlying legacy databases by inefficient treatment of single records at object layer. If an access layer is to be used for any kind of batch processing this has to be analyzed very thoroughly. If the analysis shows that the

---

[6] SQL statements of the kind"*update .... where*" are denominated as Mass Updates. One statement is able to manipulate large sets of records.

access layer will be too inefficient for such a task one should consider bypassing the access layer and coding embedded SQL statements straight into object methods. This requires special attention with regards to object identity and transaction integrity. Access layers are best suited for dialog processing. The typical work sequence here is selecting an object from a listbox, manipulating it and committing changes to the database. So in most cases only one or a few objects are written.

Batch processes on the other hand should be executed as close as possible to the original database. This means also executing them on the database server and best not on a remote client. In case batches are suitable for processing with an access layer (no mass updates or only mass updates that can be coded into isolated object methods), a second instance of the access layer stack can be compiled and installed on a central database host [Col+95].

2.6 Further Reading

It is not surprising that there is no such thing as a monograph dealing with the subject of this article. Partial aspects are treated in numerous articles. Some authors discuss the situations when best to use relational and when best to use object-oriented databases [Bur94, Kim95, Sto94]. Persistence mechanisms can be classified by their storage mechanisms and preferred use of objects (complex versus simple) - see [Kim95, Sou94]. The problem of how to integrate relational and object-oriented database technology is being discussed in many articles e.g. [Bur94, Kim95, Gra95] to quote a few. Mapping objects to relational databases has also been discussed extensively. Sample sources here are [Hah+95, Kel94b, Pre+94]. Architectures for access layers have been published in [Hah+95, Kel+95, Lip95, NeXT].

3 Product Categories

There are numerous products on the market that can be used to implement parts of the integration frame described above. This section will facilitate the search for products by giving an overview of existing product categories. Some components that are needed for an individual legacy data integration project might already be implemented in your enterprise. They must be identified and can then be reused for object-oriented data integration. We are not able to give even a nearly complete list of products here. The market for integration and middleware products is moving too fast. This is why we concentrate on product categories and give some prominent product examples. But we do not claim to be able to enumerate even a significant portion of all possible products.

3.1 Remote Database Access Products

RDA (Remote Data Access)- Products [Orf+94] enable access to a remote relational database over a network. They allow a client to execute SQL-statements on a remote server. RDA products can be used to implement parts of a tuple layer. RDA products will seldom offer access to more than one vendor's database at a time or in parallel. They are rarely suited for distributed transactions or even access to hierarchical databases.

3.2 Objectified Relational Databases

Objectified relational database products will offer an object-oriented view[7] of a relational database system. Typical objects that are offered by such class libraries are relation, query, view or similar terms. Most of these frameworks need an additional RDA product using dynamic SQL to provide their services. These products are also suited to implement parts of a tuple layer. Typical products that fall into this category are e.g. Rogue Wave's DBtools.h++ [www.roguewave.com] or a similar data access framework by Taligent [Cot+95].

3.3 Federated Databases

Federated database systems will provide a unified view of several, physically independent databases [Kim95]. They offer the user an illusion of using only one database system while using multiple databases in parallel. IBMs DRDA architecture [Orf+94] implements aspects of a federated database system. Federated databases might have to unify different SQL dialects while offering their own SQL interface. Some systems also allow integration of hierarchical data (DRDA). Federated systems can again be used to implement parts of a tuple layer. As product examples we can quote IBMs DataJoiner [www.software.ibm.com-/data/dbtools/datajoin.html], , UniSQL/M [www.unisql.com] or IBMs DRDA architecture in general [Orf+94].

3.4 Object-Oriented Databases with Relational Gateway

A few object-oriented database systems offer a so-called relational gateway to their products. This gateway is usually implemented by installing a special storage manager that replaces the normal storage manager for certain objects of the database. Many vendors offer adapted storage managers as tailored project

[7] Typical objects that are presented to the libraries users are e.g. Table, Row, Query, etc.

solutions in addition to their OODBMS. It should be straightforward to tailor such a storage manager for an arbitrary tuple layer, as long as it has been possible to implement it for any reasonable tuple interface.

An OODBMS plus relational gateway could be a turn-key solution for our integration problem. This should not lead to euphoria as problem points like integration of hierarchical database systems, reengineering of existing data resources, problems of object identity and the coupling with host transaction systems have to be checked thoroughly. Classical host environments can seldom be supported.

Product examples that fall in this category are ONTOS [www.ontos.com] or Hewlett Packard's Odapter (object adapter) products for their OpenDB architecture.

3.5 Object-Oriented Access Layers

There is also a category of products that offers the programmer an interface similar to that of an OODBMS but exclusively uses external database systems (or better data sources) as storage mechanisms. These systems do not have their own low level storage management component. The critical points to look at are again support for host databases (like IMS and DB2) and the question of collaboration with transaction systems like IMS or CICS.

Some typical products that can be named here are Persistence [www.persistence.com] or NeXT's Enterprise Objects Framework [www.next.com].

3.6 Object/Relational Databases

Besides OODBMS there is yet another family of database systems that claims to provide object-oriented data management - object relational databases [Kim95]. These databases expand relational databases by providing user defined data types, inheritance, stored procedures and further constructs. The differences, advantages and disadvantages with respect to object-oriented databases are e.g. discussed by Kim [Kim95]. Object relational databases have their own emerging standard - SQL3. Together with an access layer object relational database systems might be used to implement the integration model's object layer. Some products (like e.g. UniSQL [www.unisql.com]) also offer database federation at the level of the tuple interface. This can also be used to implement services of the integration model. Illustra [www.illustra.com] is another object relational product.

4 Case Study - Persistent Objects in a Large Bank

As an application example we will discuss the specific solution for the legacy data integration problem that has been implemented in a large German bank. First we will have to list the requirements relevant to the solution and the software environment that could be found. The use of turn-key solutions was made impossible by an array of factors that will also have to be named here. This resulted in a solution that will be presented in the following.

4.1 Situation, Requirements and Software Environment

The project is a part of a larger effort to introduce object technology in a large IS shop. It is one task to provide persistence for C++ objects.

The data resources created cannot only be isolated new databases. As in most large organizations, existing data resources from hierarchical or relational databases have to be used. The integration solution has to provide good decoupling of conventional software development from object-oriented pilot projects while both development tracks are using the same integrated data resources. The access to legacy data must not lead to any changes in existing applications. Even recompilation would be too expensive.

The software and hardware environment that could be found is typical for a large IS shop. A client/server concept incorporates a central MVS-host that will play the role of an enterprise server. The programming environment for this host is IMS-TM, PL/I, DB2 and IMS-DB. Clients run OS/2 and have been programmed in C before and will now be programmed in C++. Clients are clustered in LANs. Each LAN has its own array of LAN servers that concentrate traffic with the enterprise server. The client/host connection is implemented using APPC. A client programmer needs an integrated view of the enterprise's data resources. The situation can be summarized as a multi-layer client/server model.

9.600 baud telephone lines between branch offices place serious restrictions on the degree of carelessness one can afford concerning communication bandwidth. This resulted in the use of compression schemes, blocked data transfers and maximum lazy access schemes. The pilot solution for persistent objects will be promoted to an enterprise standard after some successful pilot projects.

The project's task was to create a programming interface for persistent objects in a new software development environment. This interface had to be as close as possible to the object-oriented database standard specification [ODMG93] to allow migration to off-the-shelf products later. As usual with persistent languages,

the programmer mustnot be annoyed with database details. What he or she sees are straight persistent objects.

4.2 Why Products Still Fail

It has always been our goal to avoid a custom solution. We would have preferred products as system software development is seldom ever the core business of a financial institution. This is why quite a lot of products were evaluated. This evaluation phase ran in parallel to the specification of a custom solution to use time gains by simultaneous engineering. However, the evaluation phase did not produce any products that were suitable for use in the given software and hardware environment. The reasons for that provide a good basic checklist for similar evaluation efforts:

- Most solutions do not support DB2 [Orf+94]. If the OS/2 variant DB2/2 is supported the solutions will use dynamic SQL [Sal93] in most cases. The use of dynamic SQL for central host databases is still forbidden by convention in many DB2-MVS shops. This is motivated by internal control procedures, authorization schemes and control of transaction load many of whom are based on the use of static SQL.
- No product on the level of the object layer was able to read or even write IMS-DB from our client platform OS/2.
- Most object layer products that offered a relational gateway were able to work with arbitrary legacy table schemes for object storage, inheritance and storing relations. Many products were designed for forward engineering and not for reengineering badly structured legacy data sources.
- The notion of object identity [Cat94] plays an important role in OODBMS. If products need to insert a new object identity into existing tables, they cannot be used in parallel to existing applications. Legacy applications don't know how to treat an additional field. They would have to be changed to update the new OID field. This is normally too expensive when thousands of legacy programs can be involved.
- We did not find a single product that was able to deal with data sources that run under a host transaction monitor (like e.g. CICS or IMS-TM). The special challenge that results from using such transaction systems is the different length of transactions on client and host. Each call to a host running a transaction system in transactional mode results in committing all open transactions on return of the call. This has to be mapped to long user transactions by using dirty reads and deferred updates and some further measures.

- None of the products was able to deal with multi-server client/server architectures (LAN servers plus MVS enterprise servers) and 2-phase-commit. This is a necessary long term requirement in the environment we found.

The above problems prevented us from using off the shelf products.

4.3 Our Architecture

With no shrink-wrapped solutions in sight, a custom solution had to be composed from products at hand. Figure 4 gives an impression of the solution's architecture. This solution follows the architecture already described in Section 2. We will only discuss those aspects here that had to be tailored with respect to existing software components or special requirements. We will describe the architecture following the layered model from bottom up.

Data access on DB2/MVS is done using conventional access layer modules. These are programmed in PL/I and generated from description files. The access modules offer the usual functionality of a relational access layer (read, insert, update, delete, read-multiple). The modules can also be used by non-object-oriented host applications. This alone has been an improvement compared to the old host architecture that did not incorporate a separate access layer.

Host access modules are called from the client sites via a transaction monitor (IMS/TM). No extra remote database access product for DB2 has to be installed. The price of this is some extra communication software. Coming from the client side, write operations have to be bundled in packages and are executed no earlier than at the client transaction's commit time. The bundling results in several update operations to be executed in a single IMS transaction. This can best be compared to on-the-flygeneration of a batch program. Buffering is provided by communications agents. APPC is used instead of DRDA [Orf+94] products, like e.g. DDCS/2. This may look strange at a first glance. But if *license costs + installation costs - programming effort - maintenance effort* are taken into account, this solution can be cheaper than shrink-wrapped products. This will not hold for each and every IS organization and any number of licenses - but it should be recalculated for each business case or project.

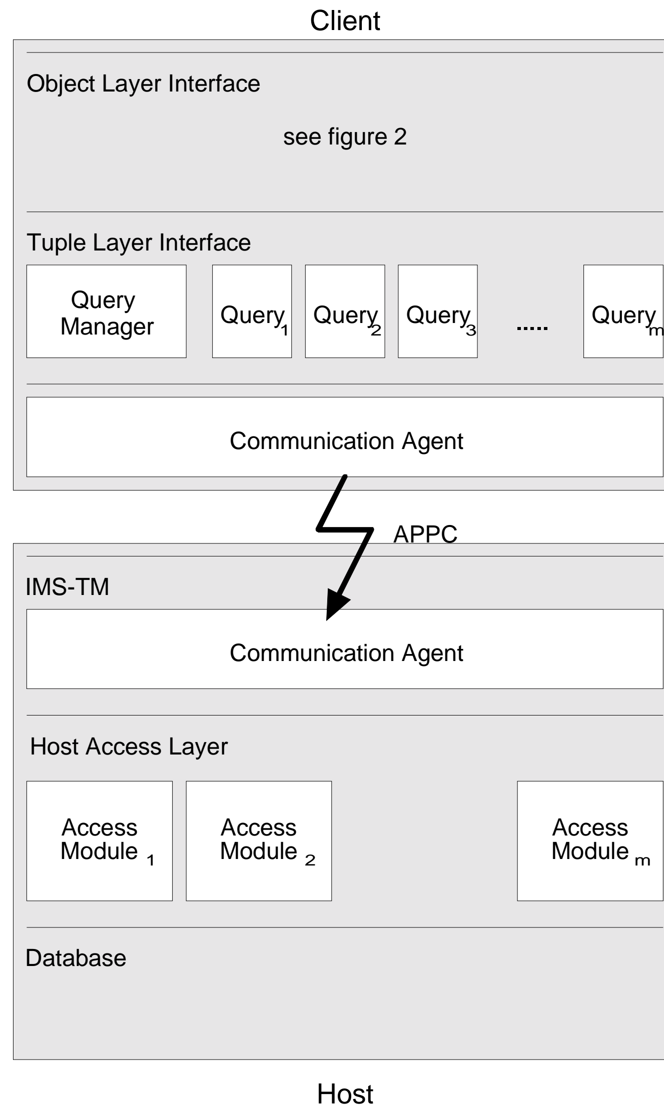

Figure 4: Project Example for an Object Data Integration Problem

A set of query objects on the client forms a tuple layer. Query objects are objectified access layer modules. In a manner of speaking, they are proxies for access layer modules on the remote server. Arbitrary queries are made possible by using a query server concept. Possible hard-coded dependencies between higher layers and queries are cut off by this server concept that can be compared to a broker architecture. Special precautions are necessary with respect to slow communication links connecting clients and server. Multiple reads can be executed using blocked transfers and reread by need mechanisms.

The object layer is constructed using the principles outlined in section 2. As before, no product could be used as most products are based on dynamic SQL as

the interface paradigm for the tuple layer. having to use an MVS enterprise server with expensive existing security and administration structures, it was not an option for a pilot project to restructure central IS procedures. Anyway, a customized storage manager project solution offered by OODBMS vendors would not have been any cheaper than the solution chosen.

A techical note at the end: Similar projects and the requirement to use static SQL have suggested the use of code generation also for C++. This is tempting at first sight but comes back as a maintenance and flexibility boomerang after some time. Runtime repositories and templates are to be preferred to code generation. This can be confirmed by a look at other architectures [Kel+95, Lip95, Wal+95, NeXT].

5 Experiences and Summary

The above architecture has produced good results in a first pilot project. It is a pity that we could not find suitable products in the first half of 1995 that fit a typical software environment for a large IS shop. As far as we know the situation has not significantly improved until now. There should be a considerable market for integration technologies, especially in large shops with a predominantly blue software environment.

Considering the very short innovation cycles in object-oriented database technology, the modularization of a solution can not be overstressed. Clear interfaces close to standards allow to exchange custum-made parts or glue with commercially available software as soon as better solutions appear on the market.

Integration products should be checked rigorously before use. The above architecture, product categories and project experiences should be helpful for an evaluation of products.

Factors that look marginal at first glance can turn out to be real expensive cost drivers. Such factors are add-ons that have nothing to do with direct database functionality, such as integration of an access layer into a security system. Classical transaction systems offer a wide range of control mechanism that may have to be reimplemented at a horrendous price.

A conventional database access layer for PL/I applications has been an important windfall profit of the project. A generator system for conventional database access modules alone can justify the rest of the costs of an integration solution.

6 References

[Atk+83] **M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, R. Morrison**: *An Approach to Persistent Programming*, The Computer Journal, 26(4), 1983.

[Bro+95] **Michael L. Brodie, Michael Stonebreaker**: *Migrating Legacy Systems, gateways, Interfaces & The Incremental Approach*, Morgan Kaufmann Publishers 1995.

[Bur94] **D. K. Burleson**: *Practical Application of Object-Oriented Techniques to Relational Databases*; John Wiley & Sons 1994.

[Cat94] **Rick G. G. Cattell**: *Object Data Management*; Addison-Wesley 1994.

[Col+95] **Jens Coldewey, Wolfgang Keller***: Objektorientierte Datenzugriffe auf dem VAA Datenmanager*, GDV Bonn, 1995 - also on Compuserve, GO CASEFORUM, File oozs.doc.

[Cot+95] **Cotter, Pottel**: *Inside Taligent Technology*, Addison Wesley 1995.

[Gra95] **Ian Graham:** *Migrating to Object Technology*, Addison-Wesley 1995.

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

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

[Kel95a] **Wolfgang Keller:** *Problems Reengineering RDBMS to Object-Oriented Databases,* Compuserve , GO CASEFORUM, File oidpr.doc.

[Kel95b] **Wolfgang Keller:** *Associations in Object-Oriented Access Layers,* Compuserve, GO CASEFORUM, File relpro.exe.

[Kel+95] **Michael Keller, Thomas Stalzer:** *Ein Erfahrungsbericht über den Einsatz von VisualAge und Vaser*, OBJEKTspektrum, Vol 2(4), Juli/August 1995.

[Kim95] **Won Kim (Editor)**: *Modern Database Systems,* ACM Press 1995.

[Lip95] **Peter Lipps:** *Enterprise Objects Framework - Fachspezifische Objekte in Open Step*, OBJEKTspektrum, Vol 2(5), September/Oktober 1995.

[NeXT] **NeXT Inc.:** Several product descriptions on NeXT Enterprise Objects Framework can be found in WWW entering from http:// www.next.com.

[Orf+94] **Robert Orfali, Dan Harkey**: *Client/Server Survival Guide;* Van Nostrand Reinhold 1994.

[Pre+94] **William Premerlani, Michal R. Blaha**: *An approach for reverse engineering of relational databases;* Communications of the ACM, May 1994, p42(9).

[Sal93] **Joe Salemi**, *PC Magazine's Guide to Client/Server Databases*; Ziff-Davis Publishing, 1993.

[Sou94] **Jiri Soukup**, *Taming C++ - Pattern Classes and Persistence for large Projects*; Addison-Wesley, 1994

[Sto94] **Michael Stonebreaker**, *Object-Relational Database Systems*, Proceedings ObjectWorld 94, London 1994.

[Str91] **Bjarne Stroustroup**: *The C++ Programming Language (2nd. Ed.)*, Addison Wesley 1991.

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

[www.xxx.com] marks references to World Wide Web sites that contain product information on products quoted in this article

White Paper

Object-Oriented Data Integration

Running Several Generations of Database Technology in Parallel

Wolfgang Keller, sd&m
Christian Mitterbauer, HYPO-Bank
Dr. Klaus Wagner, sd&m

also appeared in German, ONLINE 1996, International Congress , Hamburg, Article Number C 644

Work in Progress

Munich March 1996

sd&m
software design & management
GmbH & Co. KG
Thomas-Dehler-Straße 27
D 81737 München
Germany
Phone +49-89- 6 38 12 - 210
Fax +49-89-6 38 12 - 490