ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Rainer Manthey and Viacheslav Wolfengagen (Eds)

Advances in Databases and Information Systems 1997

Proceedings of the First East-European Symposium on Advances in Databases and Information Systems, (ADBIS'97), St Petersburg, 2-5 September 1997

Towards the Object Persistence via Relational Databases

Michal Kopecky and Jaroslav Pokorny

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN: 3-540-76227-2

Towards the Object Persistence via Relational Databases

Michal Kopecký, Jaroslav Pokorný*

Department of Software Engineering

Faculty of Mathematics and Physics

Charles University

Malostranské nám. 25

118 00 Praha 1, Czech Republic

e-mail: {kopecky,pokorny}@ksi.ms.mff.cuni.cz

Abstract

Object-oriented (OO) databases have happened important for both research and development area in the past decade. On the other hand, databases of this type have been found too expansive and not too functional in a plenty of practical tasks. New architectures appeared: object-relational data managers, relational wrapper libraries, and OO databases. The attractiveness of these solutions is that enable OO applications to be written against today's enterprise data both file- and relation-oriented. Moreover, they make OO design methodologies available for implementing business objects. Our solution of persistence of C++ object includes possibilities: to handle persistent objects via C++ programs, to conceive rows of relations as C++ objects, to use high-level query facilities in C++ programs, and to use many of database-oriented features on the level of C++ programming. The article is focused on some implementation techniques used in the GEN.LIB - a library managing a persistence through relations stored in a commercial RDBMS. The results are a part of the COPERNICUS project ADOORE.

1 Introduction

Object-oriented databases have happened important for both research and development area in the past decade. There are numerous commercial OODBMS products available on the market today. The first OODBMSs (e.g. ObjectStore) were intended to be persistent storage managers for object programming language objects. Other, more advanced ones, as e.g. Versant, O2 etc., intended as OODBMSs were realized later.

Typically, the most of the recent OODBMSs use C++ and/or Smalltalk languages as their database programming language. The application programmer may access database objects directly using the database operations in the programming language, or may perform associative lookups of objects using the query language. Unfortunately, the latter includes many problems concerning both the design of such languages and their implementation. Two significant development lines are recently at disposal: ODMG-93 standard Release 1.2 with its language OQL [Ca96] and the work proposed by the ANSI X3H2 group well-known as SQL3 that stores objects in special kinds of columns in relations. A more general solution is given by POS (Persistent Object Service) standard. POS offers a single, industry-standard interface for storing objects to any underlying datastore.

Despite of the first successes of OODBMSs in so called non-traditional applications there are still differences between many of OODBMS products, particular products suffer from various drawbacks, such as e.g. no view facilities, weak query facilities, binding to associated OO tools for analysis and design. Also any support of client/server computing is not to much supported. Shortly said, there is a lag in adoption of object-oriented databases. Another reason is that the vast bulk of the worlds data is stored in traditional datastores (not

*This Work has been supported by COPERNICUS project ADOORE (CPP94-00764).

necessarily relational), and most of the worlds programs are written under this assumption. Relational technology is still dominant and mixing the worlds of relations and objects has appeared. The idea is not new. For example in [Pr91] authors describe a technique for constructing an OODBMS from existing relational technology. They denote their resulted architecture as the object-oriented relational database. During 90ties many approaches and associated products have appeared. According to [Ha95] we distinguish among:

- *Object/Relational Data Managers* - they map objects directly to relational tables and manage objects. We count there products such as HP Oadapter, Persistence, Ontos OIS, UniSQL. For example, Oadapter can scan SQL table definitions of existing databases and figure out the corresponding C++ to make object definitions. Today, it is available as an object management level on top of Oracle.
- *Relational Wrapper Libraries* - they map objects to database objects which are linked to relational database. The relational wrapper detects a change in the contents of an object and automatically generates the SQL to make the changes in the linked relational database. Similarly, it detects changes to relational database and moves that information back into the local objects. Obviously, the translations are transparent to the user. Good examples of this approach are Smalltalk tools, e.g. VisualWorks and VisualAge.
- *Object/Relational Databases* - they store information using objects or relations, and they let developers access the data using either method. Today most important examples of this approach include last versions of traditional relational database vendors such as Oracle [Or96], Sybase [Aw97], Informix [In96], and IBM [Ch96]. They tend to so called universal server but not through the same approach.

We could also add to these trends a special functionality of some pure OODBMS that can get data form relational databases. Among well-known examples, we can cite Gemstone, O2, ObjectStore and many others. These approach use a technique of gateways in principle.

The attractiveness of, e.g., the wrapper approach is that it enables OO applications to be written today against enterprise data, making an OO design methodology available for implementing business objects [Ca96]. The GEN.LIB (General Library) system [Ko97], being implemented at Charles University in Prague, addresses many of above trends. The main goal of the library design is to make objects persistent via a relational database, that is, some data remain after a user session or application program execution. The above mentioned architectures offer more than one choice for persistence, e.g. by type, by explicit call, and by reference. GEN.LIB is based on the former possibility.

The library gives at least a possibility:

- to handle persistent and transient objects via C++ programs,
- to conceive rows of relations as C++ objects,
- to use high-level query facilities in C++ programs,
- to use many of database-oriented features on the level of C++ programming.

The GEN.LIB is part of the ADOORE¹ project. Its primary purpose was to implant the Rumbaugh's OMT methodology [Ru91] of OO analysis and design into the environment of building business applications. The GEN.LIB part of the projects provides a support for persistence of C++ application objects and some possibilities to use relational tuples as objects in C++ programs. Moreover, some higher non-procedural querying capabilities are available in GEN.LIB. Notice that there is a second ADOORE variant of the library GEN.LIB implemented via SQLWindows.

In this paper, we present some design and implementation issues of GEN.LIB. We first give a short overview of GEN.LIB (Chap. 2). Particularly, its roles in an application environment will be discussed. In Chap. 3 the GEN.LIB requirements are summarized. We then give a short description of main part of the GEN.LIB architecture (Chap. 4). The main GEN.LIB implementation aspects and decisions are contained in Chap. 5. Particularly, a typing and querying objects is discussed. Chap. 6 concludes with a summary and our plans concerning GEN.LIB usage and development.

¹Application specific Depository of Object ORiented Environment

2 GEN.LIB in an application environment - overview

GEN.LIB is a library managing a persistency of application domain objects using a relational database server. It is also able to present rows of external (by other application maintained) tables as C++ objects. Thus, GEN.LIB is, in general, a part of the client application, which provides an interface between two different paradigms: an object oriented environment on side of the client, and a relational database server on the other side. The position of GEN.LIB in an application is shown in the Figure 1.

Figure 1: Role of GEN.LIB in application

Both the GEN.LIB and the application is supposed to be developed using C++ programming language. We have chosen the ORACLE RDBMS Version 7 as the primary database server for the datastore. For OO analysis and application design of the GEN.LIB Platinum's Paradigm+ was chosen as a tool. Particularly, its support of OMT methodology is used.

3 GEN.LIB requirements

In order to respect the ADOORE project objectives (and, more precisely, the comparison of the C++ GEN.LIB implementation with the SQL Windows one), a set of common general requirements has been specified. These requirements focus on the database aspects of GEN.LIB and are listed below:

- GEN.LIB provides general relational database management facilities:
 - multiple sessions opened simultaneous on different and independent RDBMSs,
 - multiple connections simultaneously opened on the database server,
 - full support of the transaction handling,
 - full support of all RDBMS features,
 - resource allocation/de-allocation.
- GEN.LIB is a library of components that provides transparent storage/retrieval services of the objects written in the target programming language (C++) in the relational database(s). This feature includes:

- automatic and transparent SQL generation and execution,
 - enhanced memory management, transparent manipulation of the objects independent on the fact if the copy of the instance is in memory or not.
- GEN.LIB allows the execution of SQL treatment on the server side (for large set of data manipulation).
- GEN.LIB provides both object and set of objects access and manipulation facilities:
 - mapping strategies for the storage of the native C++ objects in the relational database,
 - mapping strategies for the representing data stored in the relational database using C++ objects,
 - several kinds of obtaining sets of objects using various selection criteria,
 - support for standard binary relations (one to one, one to many, many to many), retrieving objects related to given instance, etc.,
 - various locking strategies.
- GEN.LIB handles the object's instances in the memory of the client application:
 - uniqueness of each object of the domain layer,
 - format transformation between objects attributes and tables' column.
- GEN.LIB supports DB errors handling:
 - error capture,
 - exception raising,
 - strong default error handling (reconnection procedure, non-fatal error distinction, ...)
- GEN.LIB provides journalising facilities:
 - DB statement journalising,
 - DB error journalising,
 - GEN.LIB internal error journalising,
 - debugging information tracing (DB command/ statement, main classes' methods calls).

We must emphasise that this set of functionality constitutes a compromise between all the needs of the future applications that will be developed using GEN.LIB. As a consequence, an important requirement of GEN.LIB is its flexibility: the analysis of the library as well as its design should take into account the possible and reasonable extensions foreseen by its future users.

4 GEN.LIB architecture

Besides of the SQL standard, commercial RDBMS differ each other in the syntax of some commands as well as in the communication protocol between the application and the SQL server. The independence of the GEN.LIB on the used RDBMS is achieved by following of the two principles:

- First, the database dependent actions were identified during the specification phase.
- Second, the database dependent action was split into two pieces and the necessary minimum of the code was located in the separate self-standing service during the design phase.

This splitting of the services allows a more efficient future porting of the GEN.LIB from the ORACLE server to different database engines.

According to above mentioned principles we split the library into two main modules according to the database dependency of the source code.

The first module is database dependent. It means, that its code must be rewritten each time the programmer wants to access a different database server.

The second module is database independent. It uses the services provided by the lower laying database dependent part for the database accesses instead of a direct using of an concrete API and/or Embedded SQL communication.

The above described situation is shown in Figure 2.

Figure 2: Main GEN.LIB modules

The Database Independent Module covers most of the functionality of GEN.LIB using low-level services provided by the Database Dependent Module for the communication with the database server. According to required functionality of the GEN.LIB library, it is useful to divide Database Independent Module into four co-operating sub-modules. Each of them has its own role in the GEN.LIB architecture. Internal structure of the GEN.LIB including view inside the Database Independent Module is shown in Figure 3. The figure depicts also the communication paths between particular GEN.LIB modules. Let's follow the Figure 3 in the bottom-top direction:

Database Dependent Module

This module communicates with the database server using SQL commands which are send to the server through its API. The implemented Oracle communication uses OCI (Oracle Call Interface) for the communication. The Database Dependent Module also gets data from the database server and provides them to the rest of the application.

Buffer Module

The Buffer Module is the lowest part of the Database Independent Module. It is also the only module of the GEN.LIB which isn't visible directly from outside of the GEN.LIB. Its main role is to provide enhanced memory management necessary for the transparent persistent object handling. This module keeps track about all the memory copies of the persistent objects currently presented in the memory. If this module get a request asking memory location of some persistent object, the buffer either finds the appropriate copy or ensures the creating of the asked copy in the memory. When the memory is full, some of the unnecessary copies of the

persistent objects are automatically disposed. Beside the achieving transparent access to the persistent objects and uniqueness of the memory copies of persistent objects the implemented buffer speeds-up the repeated access to the objects.

Through the Interface Module the application can control the behaviour of the buffer. According to the situation, the buffer can be set either to the immediate or deferred update propagation. To ensure the continual presence of particular object in the memory the objects can be individually locked in the memory. It is also possible to override the default update propagation strategy by the individual one.

Figure 3: GEN.LIB modules - a detailed view

Persistent Object Module

The persistent object module defines the top of the hierarchy of the persistent classes and the special class for referencing its instances independently on the fact if there is a copy of the particular object in the memory or not. Those classes implement automatic SQL statement construction for the inserting, updating, deleting and querying the instances of the appropriate class.

Query Module

The main responsibility of the Query Module is to allow the access to the stored objects according to values of their attributes. In addition to this functionality, this module maintains the binary relations between classes, because obtaining all instances associated with a particular instance means obtaining all instances satisfying some condition about its contents.

The usage of some SQL query is for example the only way how to obtain first data from the database at the start time of the application, because no pointer to stored data is known to the application. Having at least one object (or at least a pointer to it) known, the application can start the usual process of spreading activity by traversing from object to object using associations between the instances of the object classes.

Interface Module

Interface Module supports the communication between the application and GEN.LIB library. It consists of only one defined solitaire class *GenLibInterface*. The application developer can usually choose, if he uses public services provided by the objects belonging to other modules directly, or if he uses services concentrated on one place.

5 Implementation of GEN.LIB features

In this section we will discuss some features of the GEN.LIB more deeply.

5.1 Database independence

As we mentioned in the chapter 4, the dependent code is separated in a stand-alone Database Dependent Module. This module implements three classes: *Cursor*, *DatabaseConnection* and *Database*. Those classes represent abstractions of the database cursor, of the connection to the database server and of the database server itself. The object-oriented approach in the Database Dependent Module design is very helpful. It allows very efficient modification of the module to communicate with another RDBMSs or even with more different servers simultaneously. For each of the used RDBMS family, one descendant of each of the above mentioned three classes should have been defined. Those descendants must implement the database dependent code for a communication with the corresponding database server. In our implementation of the GEN.LIB the classes *Oracle7Cursor*, *Oracle7DatabaseConnection*, and *Oracle7Database* for accessing of the Oracle7 database servers are implemented. The described internal structure of the Database Dependent Module is shown in Figure 4².

5.2 Supported Types of Object/Relation Mapping

The GEN.LIB requirements asked both the mapping of the C++ objects into a relational database, as well as the reverse mapping of rows of the existing regular tables into C++ objects. In addition, the possibility to execute any SQL SELECT statement on the database server and represent its result through C++ objects was required.

This situation is reflected in the internal structure of the Persistent Object Module. We introduce three generic persistent classes. The resulted solution supposes, that each derived class redefines set of the virtual string functions which return the name(s) of the selected column(s), the name(s) of the table(s) holding the data in the database and so on.

The *DatabaseObject* class is the most general of the introduced persistent classes. Its descendants derived in the application are used to hold rows of the result of any SELECT statement including GROUP BY and HAVING clauses. Instances of those classes are not really persistent - an updating their content causes no

²The graphical representation is based on Rumbaugh notation [Ru91] used in Paradigm+ tool

Figure 4: Database Dependent Module

update in the database, because the values of the different attributes can be selected from different tables or even be computed.

The *PersistentObject* class derived from the *DatabaseObject* class redefines also the database communication protocol. Instances of the classes derived from those one are supposed to hold the rows of ordinal relational tables. One class can hold data selected from only one table. Tables can have complex primary keys consisting of any number of columns of any type, but using of the GROUP BY and HAVING clauses, as well as the using of computed values in the SELECT clause is not allowed. Those restrictions allow us to propagate changes of instances in the memory back to the database.

The *OidBasedPersistentObject* class derived from the previous one provides the mapping protocol which allows to store regular C++ objects in the relational database. In addition to its parent it uses only one numeric primary key called OID (Object Identification). It is allowed to build a hierarchy of descendants of the *OidBasedPersistentObject* in the application, but GEN.LIB itself doesn't support multiple inheritance of the persistent classes.

5.3 Persistent Object Prototypes

Because the knowledge necessary to map objects to a relation is implemented using virtual methods and because C++ language doesn't allow to use virtual static functions which could be executed without having any instance of the class, we decided to require to have one empty constant instance of each derived class in the memory. This special instance is called prototype, and each instance (including prototype itself) can return its address. We propose to name this prototype using the name of the class followed by the suffix '_class'. So the prototype of class `Workers` can be declared as

```
const class Workers Workers_class;
```

Except the address of its own prototype each instance knows the address of the parent prototype. It is possible to follow the chain of prototypes of each class from its own prototype up to the root of the prototype tree.

5.4 Referencing of the Persistent Objects

It is necessary to be able to refer to any particular instance of the persistent object independently on the fact if it has its copy in the memory. It is therefore not possible to use the standard C++ pointers for this purpose. We introduce the special *DatabasePointer* class. The database pointer holds values of the primary key columns of the referred object, the pointer to the database connection which should be used to retrieve object from the database and a pointer to a prototype of the instance. Each time this pointer is de-referenced, the database pointer is translated to the memory address inside the object buffer. If no copy of the object is found, the prototype creates new instance of its own class in the memory and fulfils it with data retrieved from the database. The presence of the prototype address in the database pointer causes, that database pointers are "typed" - the database pointer points to the object of the same class as the class of the prototype. Database pointers are similar to *SmartPointers* described in [CI94].

It is possible to send all database related messages like *Update()*, *Delete()* etc. to the database pointer directly instead of sending the same message to the referred object. When necessary, the database pointer re-send the message to the object. Some of the messages can be resolved at the database pointer itself more efficiently. If the pointed object isn't in the memory, it is not necessary to update it or to retrieve the object from the database to delete it. Because the similarity between the behaviour of the persistent objects and database pointers both are derived from one common predecessor - from the *ObjectReference* class. In the similar way, the database pointers inherit the behaviour of the database objects, also the database objects can act as pointers. Each instance of database object simply point to itself.

The described situation results in the Persistent module schema Figure 5.

5.5 C++ Objects Mapping

The C++ classes can be derived one from another creating multi-level hierarchies. The hierarchy of C++ classes derived from the *OidBasedPersistentObject* class must be mapped to the relational tables. The GEN.LIB support only tree-like hierarchies, i.e. a multiple inheritance of the persistent C++ classes is not allowed. Added attributes of each of derived class are mapped to its special table. To obtain all values of all attributes of some particular derived class is necessary to select them from the join of all tables associated with that class and with all predecessors. To ensure the correct join each object inherits the *_OID* attribute defined in the *OidBasedPersistentObject* class. Each associated table has one additional numerical column named *OID*.

5.6 Querying Objects

The important feature of any library maintaining persistent objects trough relations is possibility to find objects satisfying a particular condition about its contents. Each instance of each class can generate the SQL *SELECT* statement which selects all instances of the same class. The generated SQL statements can look like:

```
SELECT    DEPT.DEPT_NO, COUNT(EMPLOYEE.OID)
FROM      DEPT, EMPLOYEE
WHERE     DEPT.DEPT_NO = EMPLOYEE.DEPT_NO
GROUP BY  DEPT.DEPT_NO
HAVING    COUNT(EMPLOYEE.OID) > 5
```

in case of descendant of the *DatabaseObject* class, or

```
SELECT    OID_ROOT.OID, EMPLOYEE.OID, EMPLOYEE.NAME,
          MANAGER.OID, MANAGER.OFFICE_NR
FROM      OID_ROOT, EMPLOYEE, MANAGER
```

Figure 5: Persistent Module

```

WHERE   (OID_ROOT.OID=EMPLOYEE.OID) AND
        (EMPLOYEE.OID=MANAGER.OID)

```

in the case of descendant of the *OidBasedPersistentObject* class.

The query in GEN.LIB is an instance of the *Query* class which consists of two strings. First string contains a fragment of the WHERE clause of the query and the second one contains a text of the ORDER BY clause. Either of those two strings can be empty. The query can be executed on any instance derived from the *DatabaseObject* class. The usual way of executing query is using the prototype of the particular class. The resulted SQL statement is constructed in the similar way as the SELECT statements shown above. Differences are only in the WHERE and ORDER BY clauses. The WHERE clause must contain the additional condition and ORDER BY clause must be added, if it is present in the query..

For example, if the query will be constructed to contain strings "EMPLOYEE.NAME LIKE 'A' statements will be (the added parts of SQL statements are underlined)

```

SELECT   DEPT.DEPT_NO, COUNT(EMPLOYEE.OID)
FROM     DEPT, EMPLOYEE
WHERE    (DEPT.DEPT_NO = EMPLOYEE.DEPT_NO) AND
        (EMPLOYEE.NAME LIKE 'A%')

```

```

GROUP      BY DEPT.DEPT_NO
HAVING     COUNT(EMPLOYEE.OID) > 5
ORDER BY   2 DESC
and
SELECT     OID_ROOT.OID, EMPLOYEE.OID,
            EMPLOYEE.NAME, MANAGER.OID,
            MANAGER.OFFICE_NR
FROM       OID_ROOT, EMPLOYEE, MANAGER
WHERE      ((OID_ROOT.OID=EMPLOYEE.OID) AND
            (EMPLOYEE.OID=MANAGER.OID)) AND
            (EMPLOYEE.NAME LIKE 'A%')
ORDER BY   2 DESC

```

respectively.

The result of the query is accessible through an instance of *QueryResult* class. This class allows the application to obtain database pointers to all instances of the result. If the query is executed on the instance of the particular class, the obtained pointers points to instances of the same class. The possibilities of working with the polymorphic collections are discussed in Section 5.8. Constructing of SQL statements for obtaining related objects (see Section 5.7) requires modification not only of WHERE and ORDER BY clauses of the statement, but also modifications of other clauses. The SQL statement construction is made in the similar way as in the case of the regular query. The additional information is stored in attributes of derived *ComplexQuery* class. The *Query* and *QueryResult* classes are shown in Figure 6.

Figure 6: Query module - Queries

5.7 Relationships Between Objects

The GEN.LIB library support all three basic types of binary relationships between persistent objects - one to one, one to many and many to many. A relationship is represented as the special instance of the *Relation* class. This class allows to add and to remove relationships between objects, to test if two objects are related, to obtain all objects related with the given instance and many others. Differences between different cardinalities of relations are solved using class specialisation as shown in Figure 7.

5.8 Maintaining Polymorphic Collections

The inheritance and polymorphism belong to native features of the OO programming. Let suppose to have the class *Employee* with two specialised descendants *Manager* and *Worker*. Sometimes it is necessary to search all

Figure 7: Query module - Relations

employees of age fifty or more. The result of such query may contain instances of both derived classes. It is possible to access all instances in the same way as instances of the class *Employee* while the sent virtual messages will be interpreted differently according to the type of particular target instance.

The situation is quite different in the case of the persistent classes in the GEN.LIB library. The obtained query result will consist of database pointers to class *Employee*. If the object is accessed via this database pointer, the accessed instance is an instance of the type *Employee*, i.e. only the part of the whole instance is retrieved from the database. It is satisfactory under some circumstances, but not every time.

One of the possible solutions is to implement the library to retrieve the correct subclass in any time the data must be retrieved. This solution would require two database accesses to complete this task. First, the type of the object must be selected, and second, the object data must be retrieved.

Finally, we chose the second solution. In this solution the database pointer can be "type-casted" to point to correct class instead of the original type. During type-casting operation, the pointer finds out the correct class of the object it points to, and change the address of the prototype from its original value to the correct one. Once the pointer is type-casted, it can be used multiple times without any additional overhead.

The class name of each C++ instance is represented as the name of the database table associated with this instance. The database table name is stored in time of inserting object in the CLASS_NAME column of the OID_ROOT table, which is associated with the *OidBasedPersistentObject* class. This column is neither selected nor updated during the work with the object itself. Having the class name (the name of the associated table), the address of the prototype must be found in the memory. To solve this problem, the binary tree of prototype addresses ordered by the names of associated tables is built at the beginning of the program during the registration phase. Each prototype should be registered using *RegisterClass()* method during this phase. The class hierarchy and the corresponding binary tree of registered prototypes are shown in Figure 8. The names of database tables in the right half of the figure are used instead of the pointers to prototypes.

6 Conclusions

The GEN.LIB system provides an integrated approach to the persistency of C++ objects, OMT methodology, and usage of exiting relational data. In the paper, we have described the architecture of the GEN.LIB and discussed some its functionality.

The existing GEN.LIB prototype will be used to build a workflow editor and simulator system WISE [De97] conceived as a BPR tool. This project is, similarly as GEN.LIB, a part of the ADOORE project. An interesting direction in the development of software tools such as the GEN.LIB is evaluating its performance in different application environments. Many performance benefits of today's relational database engines may be lost in the

Figure 8: Tree of registered classes

object-to-relational mapping layer that must exist between the objects and relational tables. We believe that workflow management could be an appropriate application for these purposes since no big volumes of data are transmitted during a workflow session.

The main GEN.LIB contribution to the "open" object world is to provide a basic framework to object persistency and using relational databases. Appropriate modifications of the Database Dependent Module can result into possibilities of connection to other relational DBMS. Another useful development if the GEN.LIB is to make it compatible with ODMG-93 standard since its object model is very similar to that of ours.

From the software engineering point of view, GEN.LIB does the first level of components designed for reuse. With GEN.LIB other class libraries and/or frameworks of e.g. business objects could be constructed. The WISE software is specified exactly in this way.

Acknowledgements

The authors gratefully acknowledge the assistance of Josef Vařica from DCIT Praha (the member of the ADOORE Consortium), who participated in a number of helpful discussions during the GEN.LIB specification.

References

- [Aw97] Agarwal, B., Wu, Daniel: *Secrets of Sybase SQL Server*. Database Programming & Design, Vol. 10, No. 5, pp 38-43.
- [Ca93] Cattel, R.G.G. (Ed.): *The Object Database Standard: ODMG-93 (Release 1.2)*. Morgan Kaufman Publishers, 1996.

- [Ca94] Cattel, R.G.G.: *Object Data Management*. Revised Edition, Addison-Wesley Publishing Company, 1994.
- [Ca96] Carey, M.J., DeWitt, D.J.: *Of Objects and Databases: A Decade of Turmoil*. Proc. of the 22nd VLDB Conference, Bombay, India, 1996.
- [Ch96] Chamberlin, D.: *Using the New DB2: IBM's Object-Relational Database System*. Morgan Kaufman Publishers, 1996.
- [Cl94] Cleal, D.: *Optimising Relational Database Access*. Object Expert, March-April, 1996.
- [De97] Dědeček, D., Dušek, L., Fedačko, V., Gatnar, P., Kutmon, T., Pavelka, J., Štourač, D., Wolny, J.: *Specification Report*. ADOORE Consortium, DCIT, Praha, 1997.
- [Ha95] Harmon, P.: *Tutorial 7. ObjectWorld Conference*, Frankfurt, Germany, October 9, 1995.
- [In96] INFORMIX: *Informix and Illustra Merge to Create Universal Server*. White Paper, February, 1996.
- [Ko97] Kopecký M., Pokorný, J.: *Objects through Relations: the ADOORE approach*. Proc. of Sixth International Conference Information Systems Development (ISD'97), Boise, Idaho, USA, 1997. (in press).
- [Me94] Melton, J.: *The "What's What" of SQL3*. Database Programming and Design, Vol. 9, No. 12, 1996, pp. 66-69.
- [Or96] ORACLE: *Introduction to Oracle 8*. Server Technologies WW Marketing Summit, September, 1996.
- [Pr91] Premeriani, W.J. et al: *An Object-oriented Relational Database*. CACM, Vol. 33, No. 11, 1990, pp. 99-109
- [Ru91] Rumbaugh, M., et al: *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [Se96] Sessions, R.: *Object Persistence: beyond object-oriented databases*. Prentice Hall PTR, New Jersey, 1996
- [St96] Stonebraker, M.: *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufman Publishers, 1996.