Semantic Data Management for db4o

Moira C. Norrie, Michael Grossniklaus, Corsin Decurins,
Alexandre de Spindler, Andrei Vancea, and Stefania Leone

Institute for Information Systems
ETH Zurich
8092 Zurich, Switzerland
{norrie,grossniklaus,decurtins,despindler,vancea,leone}@inf.ethz.ch

Abstract. Object databases such as db4o provide a very simple and effective way of making application objects persistent. However, they offer limited support for high-level database application programming in terms of facilities for the management of complex interrelated collections of data objects over long periods of time. Concepts of semantic data models such as role modelling, and associations are lacking and this results in a new impedance mismatch between the program and data models of the application. To address this problem, we have developed a semantic data management layer for db4o which supports role modelling, associations and a declarative query language.

1 Introduction

Object databases such a db4o [1] provide a very simple and effective way of making application objects persistent. They also support various forms of querying over these objects such as query-by-example mechanisms to retrieve objects, query expression trees and iterator query methods. For applications that simply require persistence of program data, these solutions avoid the well-known problems of impedance mismatch and the heavy mapping and administrative overheads often associated with the use of relational storage. However, commercial object databases lack support for high-level database application programming where facilities are required not just for the persistent storage and retrieval of data, but for the management of complex interrelated collections of data over long periods of time. Concepts well known from database systems such as constraints and triggers are usually absent and support for object and schema evolution minimal at best.

One of the major motivations within the database research community for moving to object-oriented database technologies from relational technologies was the lack of semantic expressiveness of the relational model. In relational databases, a single construct is used to represent both entities and relationships and the concepts of primary and foreign keys represent, not only relationships between entities, but also links between tuples representing parts of deconstructed entities resulting from normalisation and means of representing entity type hierarchies. A great deal of research effort in past years went into developing semantic object data models capable of not only dealing with complex structures, but

also representing object role modelling, various forms of constraints and treating associations as first class constructs. However, although numerous research systems were built based on such models (e.g. [2–4]), they were considered mainly as prototyping or modelling tools and few of these concepts have made it into commercial object databases. One of the main reasons was that, although the impedance mismatch between the program and storage models disappeared, another one was introduced between the data and the program models since the former typically required a much more flexible type system than that found in object-oriented programming languages such as Java, C# and C++.

Recently, there has been a revival of interest in object databases, in part due to the need for lightweight solutions in the rapidly growing market of mobile and embedded applications. We therefore think that it is time to revisit the issues of how to support complex data management based on object databases. In the Avon project [5], we have developed a semantic data management layer on top of db4o in order to raise the level of the application programming interface for the developers of information systems. We believe that the strength of a platform lies in the strength of the model that it supports and we therefore have based this layer on an existing semantic object data model that integrates features of extended entity-relationship and object role models with object-orientation [6].

In Sect. 2 we discuss the requirements of such a framework in terms of data modelling support. The main concepts of the OM data model on which the semantic data management layer is based are introduced in Sect. 3. The architecture of Avon is described in Sect. 4. Concluding remarks are given in Sect. 5.

2 Requirements

Object-oriented programming languages such as Java, C# and C++ model application domains in terms of object classes that specify the set of stored properties and methods used to represent application entities. Class hierarchies represent the classification of entities into semantic groups and subgroups according to the specialisation and generalisation of application concepts. When an object is created, it is created as an instance of its most specific class. In most object-oriented programming languages, the association between an object and its class is fixed at the time of creation and is exclusive. Dependencies and interactions between objects of the same class or different classes are typically represented as attributes where the values are object references.

In database application programming, data is long-lived and an object is a representation of a real-world entity throughout its lifetime. Real-world entities can have multiple roles and these roles may change over time. Query and update operations may address specific object roles. The means of representing the classification of entities therefore needs to be flexible enough to allow an object to be associated with more than one role at the same time and to gain and lose roles dynamically. For example, a newly created document may be active and a draft, then cease to be active only to later be reactivated and a final version produced. At the same time, a document about database programming languages may be

classified according to the content as a programming languages document and then later also classified as an information systems document.

Further, the dependencies and interactions between objects are role-dependent and may themselves be specialised over role hierarchies. Thus a general association between documents and conferences may be specialised into *Submitted* for the relationship between drafts and conferences and *Published* for the relationship between final versions and conferences. Relationships between objects are regarded as two way associations that can be navigated in both directions. Query languages are generally declarative and based on operations over bulk structures. It is therefore convenient if both object roles and associations can be manipulated directly through operations on bulk structures rather than having to use iterators and pointer chasing. As an example, consider a query to find all documents which were rejected at least once before being published. Ideally, we would like to express this in terms of a simple difference operation of the form *Submitted* − *Published*, taking the documents of the resulting set of pairs representing the association to obtain the required set of documents.

From the above discussion, we can identify some key data modelling requirements for database applications. We now consider each of these in turn and what it implies for an object database system.

Role Modelling

Since entities may have several roles simultaneously and different roles are associated with different object classes, *multiple instantiation* must be supported at the level of the type system. This means that objects may have multiple types where these types may belong to different paths within the type hierarchy. Further, to support object evolution, it must be possible for objects to gain and lose types dynamically. These requirements are in direct conflict with the type systems of most object-oriented programming languages which is a well-documented issue, see for example [7–11]. Several solutions have been proposed which rely on creating explicit associations between objects and role-based properties rather than relying on the type system to represent these (e.g. [10, 12]). This means that either the data management layer effectively re-implements type management or certain aspects of type checking are lost when it comes to dealing with roles. It also sharply distinguishes between types which represent fixed roles such as document and those which can be gained and lost such as draft. While this sharp distinction can be positive in terms of controlling object evolution, it results in a loss of uniformity in terms of how roles are handled. It also restricts schema evolution in that it is not easy to change a fixed role into a dynamic role and vice versa. In the software engineering world, this problem of object role modelling is addressed partially through the proposed use of the delegation design pattern [13] or explicit role modelling frameworks [14]. However, again, this is really a way around the problem based on splitting the fixed part of an object from the potentially dynamic role-based parts rather than an ideal solution. Without introducing fundamental changes to object-oriented programming languages such as introduced in database programming languages

such as Fibonacci [8], the tension between the data model abstractions and those of the programming language type system cannot be avoided. Our approach is to relieve the tension by introducing a two-level model that distinguishes typing from classification and deals with representation issues and programming language concepts at the lower level and data modelling concepts at the upper level. This enables database objects and types to be dynamically composed from programming language object instances at runtime in a manner that supports the flexibility required for object role modelling in a database application.

Associations

Representing relationships between objects as a separate construct has several advantages over the use of reference attributes. First, it is easier to ensure the consistency of binary associations since they are represented in a single place. Secondly, it enables associations to be manipulated as bulk structures and therefore can support both high-level operations over associations as well as navigational-style processing. Last, but not least, the use of a separate association construct encourages modularity and re-usability in applications. The proposal to make relationships a first-class construct in programming languages has also been around for a long time (see e.g. [15, 8]). Interestingly, in some object-oriented database systems, for example Objectivity/DB [16], binary associations are represented internally as separate constructs to facilitate consistency maintenance, but are modelled as pairs of reference attributes in the object classes involved in the relationship. We believe that this separation of entities from their relationships should also be visible at the application programming level for the reasons given above. Our approach is therefore to introduce an association construct at the data management level to represent relationships between application objects explicitly rather than as reference attributes embedded within classes at the type level.

Ad-hoc Declarative Querying

Object-oriented database systems stand at the intersection of the domains of information systems and software engineering and address requirements from both fields. As a consequence, the users working with object-oriented database systems can be software developers as well as database designers and administrators. Clearly, these two user groups have very different backgrounds and skills. Past and current object-oriented database systems have always put an emphasis on supporting the needs of developers such as an object-oriented data model together with the features traditionally known from software engineering. The needs of database designers and administrators, however, have often been neglected in comparison. Apart from functionality readily associated with database systems such as persistence, secondary storage management, concurrency control and recovery facilities, the presence of an ad-hoc declarative query language is maybe the most important requirement for database experts [17]. Often, members of this group of users are not skilled in object-oriented programming and

need to be provided with a high-level language that enables them to easily access and browse the content of a database as well as configuring and tuning the database system itself. The need for such languages has also been recognised by the ODMG Standard [18] that proposes OQL as an object-oriented query language to serve this purpose. Unfortunately, few vendors have chosen to implement the full functionality of OQL in their products in the past. Retrospectively, the failure to do so is often seen as one of the many factors why object-oriented database systems had difficulties to compete with their relational counterparts. Apart from an application programming interface, we therefore feel that it is important to offer a declarative object-oriented language to define, manipulate and query databases.

3 OM Data Model

The OM model was developed in the early 90s with the intention of bridging the gap between the semantic object models proposed for conceptual modelling and the data models of object-oriented database management systems. It was therefore important that it be semantically expressive and support key modelling abstractions [19, 20], while being amenable to efficient implementation. As proof of concept, over the last decade, there have been a number of implementations of the model based on different storage technologies, including native, relational and object storage systems [21, 4]. In parallel, specific OM languages have evolved to support data definition, querying, data manipulation and method implementation. More recently, a focus of the research within our group has been to use these systems as experimental research platforms to investigate how concepts to support domains such as web engineering and mobile computing can be integrated into database systems through an extension to the metamodel.

While the model and approach can be considered successful in terms of our experiences in using the systems as platforms for teaching and research, alongside the development of various applications, there are clear limitations in terms of scalability and adoption. The problems of scalability stem simply from a lack of resources to investigate in detail mechanisms for query processing, index structures, transaction management, distribution and storage management that would ensure good performance in the case of large-scale systems. These issues would need to be addressed if the systems were to be considered for release outside of the research community, but they are not the only factors that would limit the interest of application developers. Generally, developers want to work with familiar paradigms and languages and, ideally, would like an integrated programming environment with support for data management rather than having to interface to stand-alone database systems.

With the emergence of db4o, we noted that while there were many advances over earlier object-oriented database systems in terms of ease of application development, the underlying data model was the same, namely the abstractions offered by the type systems of languages such as Java and C#. We wanted to show how better support for database application programming could be

achieved by building a semantic data management layer on top of db4o. The OM model provides an ideal basis given its positioning between semantic data models and object-oriented database systems. At the same time, it allowed us to build on the efforts of the db4o development community in providing a persistent object store. It also enables us to address the needs of Java application developers directly by providing them with a framework to support database application programming within their development environment.

Having provided the background to our approach, we now introduce the main concepts of the OM data model. Essentially, the OM data model is an integration of the well-known Entity-Relationship (E/R) model [22] and object-oriented models, but it should be noted that it has both operational and structural components. It builds on the notions of application entities and relationships familiar from E/R models. However, in contrast to the E/R model where the notions of entity types or entity sets are often used interchangeably, OM introduces a clear separation between the typing and the classification of entities. This distinction is achieved using a two-level model. On the lower level, types describe the representation of entities, whereas the upper level captures the semantics of entities using collections to represent semantic groupings.

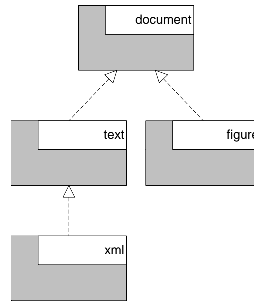

Fig. 1. OM type schema example

Each object is defined by at least one object type that specifies the attributes and methods of its instances. Object types can form type hierarchies that are built using inheritance between supertypes and subtypes as shown in Fig. 1. In contrast to most existing object-oriented systems, OM supports the concepts of *multiple inheritance* and *multiple instantiation*. Therefore while the type graph in the figure may look similar to those of typical object-oriented programming languages, a major difference is that it would be possible to create an object which is an instance of all types in that graph. This would allow us to represent the fact that an SVG document is on the one hand an XML document, but at the same time also a figure with a caption. Objects and object types are dynamically composed at run-time from information and type units, respectively, which allows this flexibility. Objects can also gain and lose types dynamically.

Objects are classified through membership in collections. As an object can be a member of multiple collections at the same time, the OM data model is said to support *multiple classification*. Each collection has a member type that governs which objects can be contained in the corresponding collection. The member type also determines the default view of objects when accessed in the context of that collection. Just as types can be specialised through subtypes, the classification of objects can be specialised through subcollections. A collection may have multiple subcollections and supercollections. Classification constraints such as **disjoint**, **cover**, **partition** and **intersect** may be placed over these collection families. For example, in Fig. 2, the collection Documents has five subcollections. On the right-hand side, subcollections Drafts and Finals are used to classify documents according to their state. Since each document must belong to exactly one of these two subcollections, a **partition** constraint is placed over them. On the left-hand side, documents are classified according to semantic content, i.e. the topic. In this case, we have only two topics and a **cover** constraint placed over them says that each document must belong to at least one of these, but it may possibly belong to both. The fifth subcollection ActiveDocuments is a collection of documents that users are currently working on.

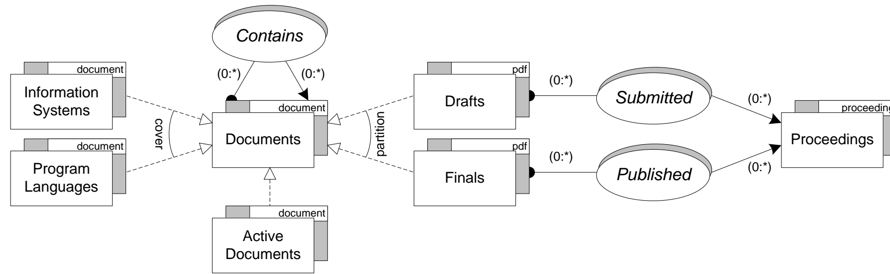

Fig. 2. Example of an OM classification schema

These five subcollections can be considered as providing three alternative classification views over documents according to status, currency and topic. A document can belong to several subcollections to represent its multiple roles. Thus a document can be active, a draft and about database programming languages and so classified under both topics. Objects can gain and lose roles by adding them to and removing them from collections. This may in turn propagate changes at the type level to ensure that the object has the correct member types. Note that since it is not required that a collection be defined for each type and it is possible to have multiple collections with the same member type, the typing layer and the classification layer are almost independent of each other.

A feature of the OM model stemming from the object-oriented programming world is the fact that collections can have different behaviours. Collections can either be sets, bags, rankings or sequences depending on whether they can contain duplicates and are ordered or unordered. A set is an unordered collection

with no duplicates. A bag is unordered but may contain duplicates. A sequence is ordered and can contain duplicates, whereas a ranking is ordered but cannot contain duplicates. All operations over collections and also the classification constraints have been generalised over sets, bags, sequences and rankings.

Relationships in OM are represented by associations that are defined in terms of a source and a target collection together with cardinality constraints. Figure 2 shows three associations—*Submitted* with source Drafts and target Proceedings, *Published* with source Finals and target Proceedings, and *Contains* with Documents as both source and target. Associations are a first-order concept of the model and are defined as n-ary collections with $n > 1$. For example, the association *Submitted* would be a binary collection with pair values of the form (d, p) where d is a member of Drafts and p is a member of Proceedings. Associations can also be specialised over roles. For example, the *Submitted* and *Published* associations could have been defined as subcollections of a general *AuthoredFor* association between Documents and Proceedings.

OM data models can be specified graphically using the notation shown in the above figures or using a textual definition language. This data definition language is a subset of the Object Model Language (OML) [23] which also encompasses a data manipulation and a query language. The query language is based on a collection algebra that defines a set of operators to manipulate and process collections and associations. Apart from being used for data definition, manipulation and querying, OML also serves as a declarative object-oriented implementation language for the methods of database objects as well as for database macros and triggers. As stated at the beginning of this section, a family of data management systems have been built that allow OM data models to be implemented directly. All systems were based on an OM metamodel and, in these systems, all data and metadata is represented as objects to allow maximum flexibility.

It is beyond the scope of this paper to describe all aspects of the OM model in detail, especially the generalised semantics of the constraints over collections and also the algebra and associated languages. Further details of the OM model can be found in [6, 4]. The remainder of the paper gives an overview over the architecture of the semantic data management layer that was developed for db4o based on this model.

4 Architecture

The architecture of the Avon framework is composed of three main layers. The lowest is the storage layer which takes care of object persistence. On top of the storage layer, the model layer implements the OM data model presented in the previous section. Finally, the interface layer maps the concepts of the model layer to an application programming interface that, similarly to JDBC, is shared by all implementations of the OM model. Thus, the semantic richness of the OM data model is made available to client applications which are required to interact with the interface layer only. Figure 3 shows the layered architecture of Avon and its main components.

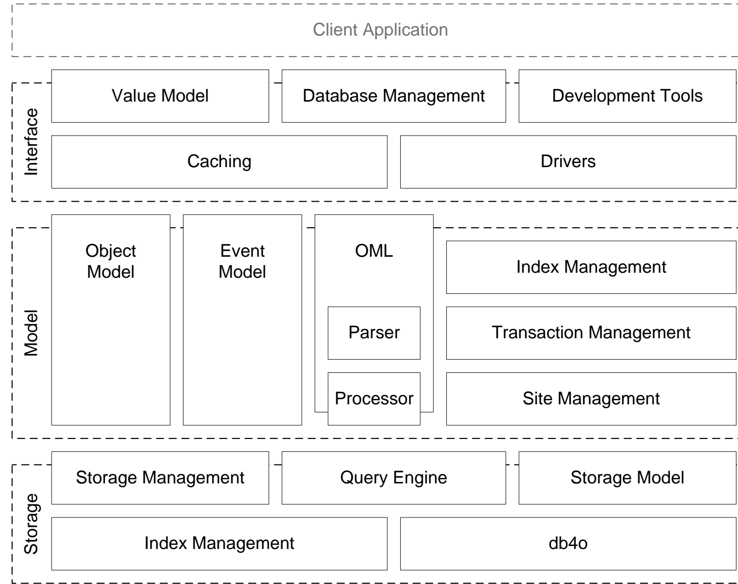

Fig. 3. Avon architecture

The storage layer encapsulates all matters of persistence by defining its own concepts for storage management and then using the db4o object database to actually store the corresponding objects. The *Storage Management* interface component is used by the model layer to insert, retrieve, update and delete objects. Data is passed to this interface component following the concepts defined within the *Storage Model* component. The *Query Engine* component of the storage layer is used by components of the model layer to post queries and get their results. It is responsible for query optimisation and processing. A low level *Index Management* component builds and maintains index structures at the level of the concepts defined within the storage model based on the index facilities offered by db4o. Also, the query engine is responsible for managing temporary objects such as collections containing query results that are too large to fit into the memory.

The concepts of the OM data model are implemented within the *Object Model* component of the model layer. Its main task is to provide a representation of the OM concepts using the Java object model. The event model features its own concepts such as event types, handlers and triggers which are pooled within the *Event Model* component. Both of these components provide the Java classes representing their concepts, therefore enabling client applications to access their functionality and manipulate the data that they hold. Consequently, together with the database interface, OML components and external events, they form the complete interface offered to client applications. The database interface methods take and return objects of the types defined in the object and event

models. The *OML* component provides the Object Model Language (OML) consisting of an algebraic query language, operation programming language as well as a data definition and manipulation language. OML scripts can be evaluated interactively using a database console or in terms of stored object methods and database macros. In both cases, the statements are parsed by the *Parser* module and interpreted by the *Processor* using the query engine of the storage layer.

The model layer also has components for index, transaction and site management. The *Index Management* component builds and maintains an index at the semantic level of the OM data model, while the *Transaction Management* component enables different concurrency control strategies to be implemented and deployed. Finally, the *Site Management* component encapsulates connectivity and communication among multiple database instances. In particular, it is used for the dissemination of events in the case that event triggers and handlers are located on different instances.

The Avon database system can be accessed by client applications through two main interfaces. One is the interface layer shown at the top of the figure and the other is the previously mentioned database console that forwards input directly to the OML component of the model layer. Based on a *Value Model* that determines the representation of all values that can occur within the OM data model, the interface layer provides a programming interface to client applications. Apart from creating, retrieving, updating and deleting objects, the interface layer also provides functionality to create, manage and delete databases through the *Database Management* module. Behind the scenes, the interface layer takes care of transparently caching objects that have been previously retrieved or created by the application to avoid unnecessary communication with the model layer. This cache is particularly important if the model and the interface layer are separated by a network. The issue of cache coherency is addressed using the trigger mechanism of the event system that allows objects in the cache to be invalidated when they change in the database. Finally, based on the interface layer, the Avon system also provides a set of development tools such as a management interface based on Eclipse and a lightweight database browser.

Figure 4 illustrates how the document concept of the application scenario presented in Sect. 3 is managed by the Avon semantic data management system. The application interfaces with the system using the Document class that offers high-level methods to access the fields of the corresponding object type. As can be seen from the figure, the class on the interface layer wraps an instance of class OMObject on the model layer.

Since the OM data model is semantically richer than the Java object model, it is not possible to map application concepts directly to Java objects. Therefore, class OMObject is the representation of an object as defined by the OM data model providing the required semantics. For example, methods dress() and strip() account for the possibility of having dynamically typed objects by allowing instances to be added to and removed from objects. Instead of high-level setter and getter methods as found in class Document, class OMObject provides generic methods to access attributes, namely setAttributeValue() and getAttributeValue().

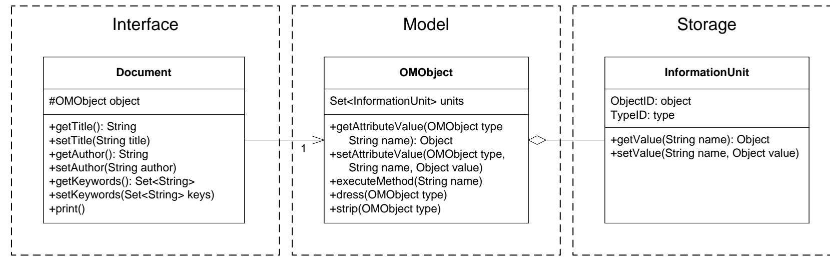

Fig. 4. Avon layers example

Instead of storing an object as a single data unit, the storage model of the storage layer uses the notion of information units represented by class InformationUnit to manage objects at the physical level. As discussed, the OM data model supports dynamic multiple instantiation and therefore an object can have a set of types that evolves over time. To cope with this requirement, each information unit managed by the storage layer corresponds to one object type and persists the values defined by this type. At run-time, the object is then dynamically composed from the instances of class InformationUnit associated with an instance of OMObject, according to the role in which the object is accessed.

5 Conclusions

We have revisited the issues of how to integrate database and programming language concepts in order to support the development of database applications within an object-oriented programming language such as Java. While seamless object persistence and querying mechanisms such as those provided in db4o are essential, we explain why they are not sufficient. Key modelling abstractions of object data models such as roles and associations as well as an ad-hoc declarative query language are necessary to represent the semantics of application entities throughout their lifetime and also support notions of modularity and reusability in application development. We describe how we have addressed this by developing a database application programming framework for Java programmers by building a semantic data management layer on top of db4o.

References

1. Paterson, J., Edlich, S., Hörning, H., Hörning, R.: The Definitive Guide to db4o. Apress (2006)
2. Missikoff, M., Toiati, M.: MOSAICO – A System for Conceptual Modelling and Rapid Prototyping of Object-Oriented Database Applications. In: Proceedings of ACM SIGMOD International Conference on Management of Data. (1994)
3. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S.: ConceptBase – A Deductive Object Base for Meta Data Management. Journal of Intelligent Information Systems **4**(2) (1995) 167–192

4. Würgler, A.P.: OMS Development Framework: Rapid Prototyping for Object-Oriented Databases. PhD thesis, ETH Zurich, Zurich, Switzerland (2000)
5. Global Information Systems Group, Institute for Information Systems, ETH Zurich: OMS Avon Project Page: http://maven.globis.ethz.ch/projects/avon/.
6. Norrie, M.C.: An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In: Proceedings of International Conference on the Entity-Relationship Approach, Arlington, TX, USA. (1994) 390–401
7. Pernici, B.: Objects with Roles. In: Proceedings of IEEE/ACM Conference on Office Information Systems. (1990)
8. Albano, A., Ghelli, G., Orsini, R.: Fibonacci: A Programming Language of Object Databases. VLDB Journal **4**(3) (1995)
9. Bertino, E., Guerrini, G.: Objects with Multiple Most Specifc Classes. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP 95). (1995)
10. Gottlob, G., Schrefl, M., Röcki, B.: Extending Object-Oriented Systems with Roles. ACM Transactions on Information Systems **14**(3) (1996)
11. Kappel, G., Retschitzegger, W., Schwinger, W.: A Comparison of Role Mechanisms in Object-Oriented Modeling. In: Proceedings Modellierung'98, Report Nr. 6/98-I (Angewandte Mathematik und Informatik, Universität Münster) (1998)
12. Supcik, J., Norrie, M.C.: An Object-Oriented Database Programming Environment for Oberon. In: Proceedings of the Joint Modular Languages Conference (JMLC'97), Linz, Austria (1997)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
14. Riehle, D., Gross, T.: Role Model Based Framework Design and Integration. In: Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98). (1998)
15. Albano, A., Ghelli, G., Orsini, R.: A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language. In: Proceedings of Very Large Database Conference. (1991)
16. Objectivity, Inc.: Objectivity/DB, Version 9.3: http://www.objectivity.com.
17. Atkinson, M., Bancilhon, F., DeWitt, D., Ditrich, K., Maier, D., Zdonik, S.: The Object-Oriented Database Manifesto. In: Proceedings of International Conference on Deductive and Object-Oriented Databases. (1989) 223–240
18. Cattell, R.G.G., Barry, D.K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., Velez, F.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann Publishers Inc. (2000)
19. Smith, J.M., Smith, D.C.: Database abstractions: Aggregation and generalization. ACM Transactions on Database Systems **2**(2) (1977) 105–133
20. Peckham, J., Maryanski, F.: Semantic Data Models. ACM Computing Surveys **20**(3) (1988) 153–189
21. Kobler, A., Norrie, M.C.: OMS Java: Lessons Learned from Building a Multi-Tier Object Management Framework. In: Proceedings of Workshop on Java and Databases: Persistence Options, November 2, 1999, Denver, CO, USA. (1999)
22. Chen, P.P.: The Entity-Relationship Model – Towards a Unified View of Data. ACM Transactions on Database Systems **1**(1) (1976) 9–36
23. Lombardoni, A.: Towards a Universal Information Platform: An Object-Oriented, Multi-User, Information Store. PhD thesis, ETH Zurich, Zurich, Switzerland (2006)