Appeared in: Zicari, R., editor, ObjectWorld Frankfurt'95 – Conference Notes, Wednesday, October 11, Software Development Track SD.6, 1995.

Object-Oriented Database Design: What is the Difference with Relational Database Design?

Gunter Saake Stefan Conrad Ingo Schmitt Can Türker

Institut für Technische Informationssysteme
Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D–39016 Magdeburg, Germany
E-mail: saake@iti.cs.uni-magdeburg.de

Abstract

Object-oriented database design is not only a simple extension of relational database design. By modelling structure as well as behaviour of real world entities as coherent units, object-oriented database design succeeds in capturing more semantics of applications already in the design phase. The use of object-oriented concepts like inheritance allows an adequate modelling and a better application implementation based on an object-oriented database system. However, the results of object-oriented design can also be applied to classical database systems.

Keywords: object-oriented database design, relational database design, semantics of applications.

1 Introduction

In this paper we clearly point out the main differences between object-oriented database design and relational database design. There is an amazing confusion about the meaning of object-oriented database design which is often mixed up with object-oriented modelling (or specification), with object-oriented analysis, or object-oriented design. Therefore, we make a strict distinction between these related object-oriented approaches.

The starting point of our discussion is database design for the relational database model [Cod70, AD93]. Usually, the Entity-Relationship model [Che76] is used for conceptual database modelling [BCN92] and then a transformation to the relational database model is necessary because there are no Entity-Relationship database systems, at least no commercial ones. For a more detailed discussion on this see any general database textbook like [EN94, HS95].

In the object-oriented area there exists a number of design approaches, e.g. [CY90, RBP+90, Boo91]. Those approaches often claim to cover the whole spectrum from requirements analysis to system design (and sometimes even up to the implementation).

In addition, they are often proposed to be ideal for object-oriented database design. However, we think that these object-oriented methodologies cannot directly be used for designing object-oriented databases because they neglect or even ignore certain properties expected in the context of databases (integrity constraints, transactions, etc.). Another object-oriented approach to conceptually modelling of information systems is a formal specification approach [JSHS95] which is based on a temporal logic framework for objects. In contrast to the object-oriented methods mentioned before it has a clear and sound formal semantics. At the time being, all these object-oriented approaches fail in providing a similar, well-developed theory like that of relational database design. For instance, there are no quality criteria available which could be used for some kind of object-oriented normalization of database designs in order to achieve the best possible design.

With respect to these object-oriented design methods we make clear what object-oriented database design should be, how it is related to relational database design, and where the main differences are. Therefore, we firstly recall the notions and issues of relational database design. Then, we discuss the benefits of object-orientation for database modelling by introducing essential object-oriented concepts. Afterwards, we conclude several consequences for object-oriented database design by comparing with classical relational database design based on Entity-Relationship modelling. To round off the presentation we briefly sketch the current state of object-oriented design methodology. Finally, we conclude by summing up and giving a short outlook to the possible future development of the area.

2 Relational Database Design: Notions and Issues

The relational data model is based on the notion of *relation* as only supported data modelling structure. The theory of relations is simple and elegant; the frameworks of relational algebra and relational calculus are built upon fundamental mathematical notions like sets, functions, and predicate logic.

Based on the mathematical structures behind the relational model we can define a database design theory using the primitives of the relational model [AD93]. Basically, we have relation schemata as collections of attribute names, and relation instances over them. Properties of relational databases must be expressed in this terms only. Examples of such properties are functional dependencies relating attributes of a relation schema or inclusion and join dependencies relating instances of different relation schemata. Based on these dependencies we can define the well-known framework of normal forms to discuss redundancies and other anomalies of a relational design.

What are the kinds of design requirements we can capture using this approach? First of all, we talk about database states as collections of relations. The dynamic of a database is not handled explicitly. Mostly dependencies are used as input for normalization algorithms to control redundancy of data. With other words, relational database design captures *aspects which can be encoded in the relational database structure* only.

For this task, relational design has a well-developed theoretical background. Normalization algorithms are formally specified and realized in database design tools; complexity and decidability results for normalization are known for the relevant kinds of dependencies. With relational theory we have the tools for analyzing redundancy in storing data.

However, this well-developed theory brings with it the danger of ignoring other design requirements. If we focus on structural redundancy only, we may forget about other constraints and database properties. The framework for relations is simple which helps to decide redundancy properties on the one hand, but hinders the adequate modelling of other data structures like lists and arrays (which are usually connected with the use of specific operations sometimes not even expressible in languages like relational algebra). Such data structures have to be modelled in an artificial way, and relational design theory will ignore the special properties of them.

Another drawback of the relational approach is the exclusion of the dynamics dimension in the design framework. Database items are not only characterized by their structure, but also by the specific functions applied to them and the temporal evolution of the stored data. Dynamics in relational database theory on the other hand is purely based on the generic operations **insert**, **delete** and **update**.

The ER approach is commonly used as an early design step in designing relational databases. ER modelling makes the structural properties handled in relational database design explicit using a graphical notation. ER schemata are therefore well suited for designing structural aspects of an information system but have to be extended to handle system dynamics if necessary. The pure ER approach has the same deficiencies as the relational approach for handling complex data structures like arrays and to handle functions in design. This deficiency is partly repaired in extended ER models covering these aspects explicitly.

> Relational database design tries to encode application semantics inside the relation type structure only. This is too restricted for most interesting application scenarios.

3 Benefits of Object-Orientation for Database Modelling

Although up to now there is no common object model, there is a general agreement (at least in the database community) that object-oriented data models provide a semantically richer framework for modelling and implementing complex information systems. In the last ten years several object models which support many different concepts were introduced in order to fulfill the demands of advanced applications. In this section we briefly recall the fundamental, common concepts of the proposed object models and point out some benefits when using these concepts for database modelling.

Integration of structure and behaviour. When modelling real world entities in terms of relations, the behaviour of these entities is always represented in corresponding application programs. In contrast, the object-oriented paradigm integrates structure as well as behaviour of real world entities into coherent units, the so-called *objects*. The structure of an object is represented by a set of *attributes* while the behaviour is expressed by a set of operations (usually called *methods*) that are applicable for that object. By integrating structure as well as behaviour of real world entities, object-oriented database design succeeds in capturing more semantic of applications already in the design phase.

Classes, types and inheritance. In contrast to most of the object-oriented programming languages (like C++), (pure) object-oriented database systems distinguish between types and classes of objects. The term *(object) type* has an intentional character and refers to definition of common properties (structure and behaviour) of a set of similar objects, whereas the term *class* describes the extensional character of a set of similar objects.

Classes and types are usually organized in *inheritance hierarchies*. In case of type hierarchy, the inheritance mechanism is used to factorize shared specifications and implementations. A *subtype* of an object type specializes the properties of that object type by redefining existing and adding new properties, e.g. a student may have additional attributes like university or faculty beside the inherited ones of a person. Contrary, in case of class hierarchy, objects are grouped into several extensions. Here, a *subclass* specializes the extension of a superclass; the objects of the subclass represent a subset of the objects of the superclass, e.g. each student *is a* person.

In general, we can state that classifying, typing and using inheritance helps to better structure and describe database schemas. Besides, (syntactical) inheritance produces compact code by reusing existing attributes and methods of an object type, and thus reduces the costs for the programming.

Composite objects. A *complex object (or composite object)* may be composed from several objects that may be composite themselves, e.g. a car is built from (among others) a chassis, four wheels, and a motor which itself is a complex object with many components.

For building composite objects, a set of type constructors like `tuple-of` or `set-of` are provided by object-oriented database systems. In general, these constructors are applicable in a orthogonal fashion. In comparison with that, the constructors of the relational model do not meet this requirement, because the only existing abstraction is the relation which is a set of tuples of atomic values. Furthermore in opposition to relational database systems, object-oriented database systems allow the transitive propagation of an operation on a composite object to all its components.

The concept of composite objects is an important means for more realistic modelling. Especially in case of modelling relationships and dependencies between objects, the benefits of this concept are obvious.

Object identity. Whereas tuples are identified in a relation by a given key, objects are uniquely distinguished and accessed by object identifiers which are independent from their attribute values. An object identifier further has the following properties:

1. it identifies an object not only in an extension of a class, but also in a database,
2. it is constant throughout the lifetime of the object,
3. it is not visible to the user, and
4. it is created and maintained by the system.

The object identity approach has several advantages. Objects can refer to other objects simply by using their object identifier. A schema designer, therefore, is not responsible for finding adequate keys for various classes of objects in order to express referential relationships. Obviously, there is no need to introduce artificial attributes, for instance, an employee or social security number in a person class.

Views and encapsulation. Following the idea of abstract data types, objects are separated in an interface and an implementation part. The interface part is the specification of a set of properly defined methods that may be used to access and modify the attribute values of an object. Thus, the interface captures the external view on an object. The implementation part, on the other hand, contains the details of an object (data as well as method implementations).

By encapsulating the internal representation of an object, a kind of logical data independence is achieved. Changes of the internal structure and/or implementation do not require the modification of applications that are using that object type. Thus, encapsulation helps to maintain complex software by providing a means for structuring systems in independent, easily changeable modules. Moreover, encapsulation can also be used for realising protection or authorization issues, e.g. integrity or access control conditions can be checked in the methods provided by the interface.

In conclusion, we may state:

Object-oriented modelling provides adequate concepts for OODB design.

4 Consequences for OODB Design

After having seen that object-orientation offers more and semantically richer concepts for modelling, we have to answer the question which are the consequences for OODB design compared with relational database design. Here, we try to find out the advantages of OODB design by comparing it with relational database design.

Typically, relational database design does not start by simply writing down a relational schema. In practice, nearly everybody begins the design on a more abstract and realistic level. Thus, finding entities (and entity types) as well as relations (and relationship types) between entities is often the first step towards a relational database schema. The result of this first design phase is usually an Entity-Relationship schema which now has to be translated into a corresponding relational schema. Of course, this can be done automatically, i.e. by a software tool. However, as we all know, the result of such an automatic translation is in general not good. The usual approach to repair the deficiencies is normalization. In the end there is often a collection of relation schemas which should be (somehow) equivalent to the original Entity-Relationship schema. Nevertheless, the result is not always a "good" relational database schema. One of the main reasons for this is that it is difficult to get an intuitive meaning of the relations without knowing about the corresponding Entity-Relationship design. Furthermore, there is a lot of semantic information inherent to the Entity-Relationship schema which cannot be found in the relation schemas. Obviously, we have lost a part of the intended semantics.

But, is object-oriented database design better at all? Or do we also lose semantics? A first provocative answer is that OODB design is not better than relational database design. But this is only correct in case we apply OODB design in a naive way, i.e. if we only use object-orientation for modelling the structure of objects and classes and forget about the dynamic behaviour of the objects. In that case there is no real improvement on the Entity-Relationship approach. However, the object-oriented approach can win by capturing more application semantics (i.e. by describing dynamic behaviour of objects) and by bringing this additional semantics into the final implementation. Only then, there is a real profit of object-oriented modelling compared with the classical Entity-Relationship based approach for relational databases.

In principle, OODB design is tailored for object-oriented database systems. Therefore, it supports all properties object-oriented database systems offers. Beside structural modelling concepts which can also be found in (extended) Entity-Relationship approaches, the possibility to describe behaviour of objects is one of the main, distinctive properties of OODB design. Nevertheless, there is an obvious gap between OODB design and object-oriented databases. Currently most commercial object-oriented database systems fail in supporting the whole range of concepts which are available in object-oriented modelling. In consequence, several concepts used for object-oriented modelling have to be translated to the concepts of the available object-oriented database. In fact, this can result in a loss of semantics because different concepts used in the design are mapped to the same concept of the database.

Often OODB design is only discussed with regard to object-oriented databases. From our point of view it can even be employed for relational databases. Of course, this is not the original intention of OODB design, but it works, and it can work even better than traditional relational database design starting with a variant of Entity-Relationship modelling. The main advantage of OODB design for relational databases is that it allows to catch more application semantics, especially by dealing with behaviour. Whereas the structural part can be translated to relation schemas in a way similar to the traditional approach, the behaviour part is not wasted. It is possible to generate triggers as well as stored procedures for the relational database system and, thereby, the behavioural semantics of applications can be brought into relational database systems.

In conclusion, we may state that OODB design is an essential improvement of traditional database design even if the target of implementation is not an object-oriented database system, but a relational database system.

The main statements of this section can be summed up as follows:

1. Object-oriented database design without considering dynamics is just another way of Entity-Relationship modelling.
2. Object-orientation offers the possibility to capture more application semantics for object-oriented databases.
3. Currently, there is a gap between OODB concepts and object-oriented modelling concepts.

5 Current State of OODB Design Methodology

In the previous chapter we compared OODB design with relational design and pointed out some advantages of an OODB design. We stated that using object-oriented concepts in the design phase allows to catch more application semantics in an early stage of the software life cycle process. The fundamental concepts of object models were sketched in section 3.

But the usage of object-oriented concepts is not sufficient enough for a good database design. Practical experiences have shown, that a designer has to be assisted and taught how to apply the powerful object-oriented concepts in a correct way in order to guarantee a good design. A high quality design is extremely relevant for the success of a whole software project. This aspect should not be underestimated.

Because of the complexity of OODB design, a sound design technique is needed to make the design of an OODB easier and better. A methodology helps to grasp the right information from the universe of discourse in the right sequence and to model it by the right concepts. But a methodology can only assist the designer. The design process is and will always be a creative and hard work, that cannot be fully automated by a machine. Especially, finding the right objects remains the main problem in the design process. At this point we can say that there exists no generally accepted methodology for the design of an OODB. In contrast, in practice several methodologies for object-oriented design are used. The most frequently used object-oriented analysis and design methodologies are OMT [RBP+90], OOA/OOD [CY90, CY91] and OOD [Boo91]. Typical design steps in these methodologies are

- finding objects,
- classifying objects to classes,
- defining the internal structure of classes (attributes),
- modelling static connections between classes (instance connections and inheritance connections),
- modelling structural connections (aggregation connections),
- fixing message passings between objects using static connections, and
- describing the internal behaviour of classes.

The steps described above cannot in general be executed sequentially. For instance, the description of the behaviour may cause to change the internal structure of the class. So during the design process, cycles in the design steps are inevitable. Therefore the term of a cyclic refining design approach is more used than of a top-down or of a bottom-up approach.

For a better control of the complexity of an object-oriented design, different kinds of diagrams are introduced by the methodologies to model different aspects of the problem area in a graphical fashion. So the structural object model describes the data structure:

classes with instance connections, aggregation connections and inheritance relationships, similar to the ERM [Che76]. Process models are used to express the behavioral aspect of classes. The process model is often based on simple or extended state-transition diagrams.

Problems during an object-oriented design process often arise from losing track. These problems increase if several kinds of diagrams and concepts respectively are not mutually coordinated in a sufficient way. If this is not done correctly, graphical presentations in form of diagrams cannot compensate this problem. Another point is the informal description of the concepts introduced by the methodologies mentioned. In general, an informal description may lead to different understandings. This can hinder a necessary exchange of design results between different persons.

Some problems arise due to the use of object-oriented modelling instead of object-oriented database design methodologies.

- The object-oriented design methodologies focus on finding application objects and describing their properties both in terms of structural properties as in terms of functional behaviour. However, the tuning of the object collection towards database requirements is outside the scope of these methods. There are no quality criteria for object database schemata measuring the degree of redundancy like it is done by the framework of normalization for the relational case. Transformation rules for transforming object schemata towards better data representation are still missing — there exists no counterpart to normalization as it is known from the relational case. However, there is research in these topics, but there is no commonly accepted framework around like it is in the relational case.

- During the object-oriented design process there exists no clear distinction between the design of application and database semantics. This avoids the problem of impedance mismatch, but in accordance with the three-level architecture [TK78] the demand for logical independence is violated. Mixing application and database semantics hinders a database system to be an integrated platform for many different applications.

- The known object-oriented design methodologies often fail to specify integrity constraints declaratively and explicitly. Instead, such constraints have to be encoded in methods and cannot be handled by the database system. In consequence, consistency of a database can only be guaranteed by the database system in conjunction with its applications.

In conclusion, we have shown the significance of a methodology assisting the object-oriented design process. The methodologies mentioned above propose different design steps. We also sketched some limits of current methodologies resulting from informal methodology description, the lack of normalization steps and the complexity of the concepts proposed.

1. Up to now, there is no own OODB design methodology.

2. Object-oriented analysis and design methodologies are only suitable for OODB design to a certain degree.

6 Conclusions and Outlook

As it is argued in the previous sections, the concept of object-orientation enables to consider more application semantics during the design process than it was possible for relational database design. However, relational database design had a lot of positive aspects which should be developed for OODB design, too. Especially, we need the following for designing OODBs in a profound way:

1. A *theoretical background* for OODB design like it was developed for relational databases.

2. A collection of *quality metrics* measuring relevant properties of an OODB design, for example, the degree of redundancy.

   In relational design, this measurement corresponds to test on normal forms. In OODB design, we will surely have several different metrics (which may be conflicting), because we make more application semantics explicit.

3. For tool support, we need *transformation rules* which can be applied to object database schemata to reach a higher degree of quality (with respect to a specific quality metrics).

   In relational databases, these transformations correspond to normalization algorithms.

In this paper, we have only sketched some concepts and problems of OODB design in contrast to relational design. Because of space restrictions, we completely ignored some other aspects, like for example new problems arising with view integration of object-oriented database schemata.

What may be a résumé of our discussion? First of all: Remember the past! Relational design theory has done a quite good job for the restricted scope it was developed for. The ideas of normalization and detection of redundancies and anomalies are still valid in the object world. Second: Use the new possibilities of object-orientation to capture more application semantics inside the database objects. But third: A coherent theory of OODB design has to combine both worlds, which is still missing and will surely need some time before it has found its way into commercial database design.

References

[AD93] P. Atzeni and V. De Antonellis. *Relational Database Theory.* Benjamin/Cummings, 1993.

[BCN92] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design — An Entity-Relationship Approach.* Benjamin/Cummings, 1992.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, Redwood City, 1991.

[Che76] P. P. Chen. The Entity-Relationship Model – Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Design*. Yourdon Press – Prentice Hall, Englewood Cliffs, NJ, 1991.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin Cummings, Redwood City, CA, 2 edition, 1994.

[HS95] A. Heuer and G. Saake. *Datenbanken — Konzepte und Sprachen*. International Thomson Publishing, Bonn, 1995.

[JSHS95] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 1995. *To appear.*

[RBP+90] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[TK78] D.C. Tsichritzis and A. Klug. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems. *Information Systems*, 3(3):173–191, 1978.